Contribution au décodage à décision douce des codes correcteurs en bloc

Senad Mohamed Mahmoud

UNIVERSITE
BRETAGNE
LOIRE

THÈSE / Télécom Bretagne
sous le sceau de l'Université Bretagne Loire

pour obtenir le grade de Docteur de Télécom Bretagne
En accréditation conjointe avec l'Ecole Doctorale Sicma
Mention : Sciences et Technologies de l'Information et de la Communication

présentée par
Senad Mohamed Mahmoud
préparée dans le département électronique
Laboratoire Labsticc

Contribution au décodage à décision douce des codes correcteurs en bloc

Thèse soutenue le 13 janvier 2016
Devant le jury composé de :

Christophe Jego
Professeur, Bordeaux INP / Enseirb-Matmeca / rapporteur et président

Ayoub Otmani
Professeur, Université de Rouen / rapporteur

Jean-Claude Carlach
Ingénieur chercheur, Orange Labs / examinateur

Michel Jézéquel
Professeur, Télécom Bretagne / directeur de thèse

Patrick Adde
Directeur d'études, Télécom Bretagne /invité

N° d'ordre : 2016telb0394

Sous le sceau de l'Université Bretagne Loire

Télécom Bretagne

En accréditation conjointe avec l'Ecole Doctorale Sicma

Contribution au décodage à décision douce des codes correcteurs en bloc

---

Thèse de Doctorat

Mention : Sciences et Technologies de l'Information et de la Communication (STIC)

Présentée par **Senad Mohamed-Mahmoud**

Département : Electronique

Directeur de thèse : Michel Jézéquel

Soutenue le 13 janvier 2016

Jury :

M. Christophe Jégo, Professeur, Bordeaux INP/ Enseirb-Matmeca (Rapporteur).
M. Ayoub Otmani, Professeur, Université de Rouen (Rapporteur).
M. Michel Jézéquel, Professeur, Télécom Bretagne (Directeur de thèse).
M. Jean-Claude Carlach, Ingénieur de recherche, Orange Labs (co-encadrant de thèse).
M. Patrick Adde, Directeur d.études, Télécom Bretagne (invité).

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui, de près ou de loin, ont contribué à la réalisation de ce travail de thèse de doctorat. Ma reconnaissance s'adresse d'abord à M. Christophe Jego et M. Ayoub Otmani, qui ont accepté d'être rapporteurs de ma thèse, pour leur lecture attentionnée du rapport et leurs remarques constructives.

Je voudrais remercier aussi mon encadrant Jean-Claude Carlach pour son implication, son soutien et ses conseils dans les différentes phases de ce travail. Je remercie également mon directeur de thèse le professeur Michel Jezequel pour son écoute et son soutien tout au long de mes travaux de thèse, et Patrick Adde pour sa sympathie, sa disponibilité et son aide.

Je salue les membres de l'équipe CREM à Orange Labs Rennes qui m'ont accueilli pour l'élaboration de ce projet. Les nombreux échanges que nous avons eu tout au long de ce travail m'ont beaucoup apporté et motivé. Je remercie particulièrement Jean-Christophe Rault pour son accueil chaleureux, son écoute et sa disponibilité.

Je remercie aussi tous ceux qui ont été mes collègues de bureau pour la bonne ambiance et les différents échanges : Jean, Marc-Antoine, Naila et Wissem.

Je remercie enfin ma famille pour son soutien régulier.

RÉSUMÉ

Pour des trames de longueurs n supérieures ou égales à quelques milliers de bits ($n > 1000$), les turbocodes et les codes LDPC (Low-Density Parity Check) ou leurs variantes donnent d'excellentes performances avec une complexité raisonnable pour des implémentations "temps réel" même pour des débits très élevés (GigaBit/s).

Par contre pour les petites longueurs ($n < 1000$) de trames, les turbocodes et codes LDPC courts sont moins performants. En effet, pour avoir de grande distances minimales d_{min} avec des codes de petites longueurs n et de grandes capacités de correction, il faut des codes dont les matrices de contrôle de parité soient de densité élevée et non creuse, c'est à dire avec beaucoup de 1 et très peu de 0, comme celles des codes LDPC ou des turbocodes. La densité élevée de la matrice de contrôle du code conduit à des cycles courts dans le graphe de Tanner correspondant et donc rend les algorithmes type BP (Belief Propagation) inefficaces.

L'objectif principal de cette thèse est donc de contribuer à améliorer le décodage à décision douce des codes en bloc linéaires courts. Nos travaux se sont orientés, dans un premier temps, vers l'étude d'une classe particulière de codes en bloc appelés codes Cortex. La construction Cortex repose sur l'utilisation de codes de base de petites longueurs assemblés en étages reliés par des permutations. Nous proposons une nouvelle méthode pour estimer les variables cachées dans la structure afin de pouvoir effectuer un décodage itératif entre les codes de base. Nous proposons une deuxième méthode de décodage qui ne nécessite pas d'estimation des variables cachées en les dissimulant dans des équations booléennes.

Dans un second temps nous avons élargi nos recherches à des algorithmes de décodage très généraux valables pour tout code en bloc. Nous proposons dans ce cadre trois méthodes de décodage basées sur des treillis-produits de complexité réduite construits à partir des lignes des matrices génératrice et de contrôle du code en bloc.

ABSTRACT

For lengths n greater or equal to several thousands of bits, turbocodes and LDPC (Low-Density Parity Check) codes or theirs alternatives achieve a very good performance with reasonable implementation complexity in a real time even for very high data rates (GigaBit/s).

However for small code lengths ($n < 1000$), turbocode and LDPC codes are less efficients. Indeed, to get short block codes with high minimal distance d_{min} and high correction capabilities, theirs parity-check matrices must be not sparse *i.e* with much 1 and little 0 like the LDPC or turbocodes matrices. The high density of the parity-check matrix yield to short cycles in the associated Tanner graph and therefore makes the algorithms type-BP (Belief Propagation) not efficiencts.

The main objective of this thesis is then to contribute to the soft decision decoding of short linear block codes. Our work was directed, initially, toward the study of a particular class of block codes called Cortex codes. Cortex construction is based on the use of component codes with small lengths assembled in stages connected by permutations. We propose a new method to estimate the hidden variables in the structure in order to perform iterative decoding between the component codes. We propose a second decoding algorithm which does not require estimation of hidden variables by concealing them in Boolean equations.

Secondly we have expanded our research to propose very general decoding algorithms valid for any code block. We propose in this context three decoding methods based on reduced complexity trellis-products constructed from the lines of generator and parity-check matrices of block code.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

AWGN : Additive White Gaussian Noise.

BCD : Binaire Codé Décimal.

BCH : Bose-Chaudhury-Hockenghem.

BCJR : Bahl-Cock-Jelinek-Raviv.

BEC : Binary Erasure Channel.

BER : Bit Error Rate.

BP : Belief Propagation.

BPSK : Binary Phase Shift Keying.

dB : décibel.

DMC : Discret Memoryless Channel.

IEEE : Institute of Electrical and Electronics Engineers.

LDPC : Low-Density Parity Check.

LLR : Log-Likelihood Ratio.

LVA : List Viterbi Algorithm.

ML : Maximum Likelihood.

MS : Min-Sum.

PLVA : Parallel List Viterbi Algorithm.

QEC : Quantum Error Correction.

RM : Reed-Müller.

SISO : Soft-Input Soft-Output.

SLVA : Serial List Viterbi Algorithm.

SNR : Signal-to-Noise Ratio.

SOVA : Soft-Output Viterbi Algorithm.

VLSI : Very Large Scale Integrated.

LISTE DES NOTATIONS

Sauf indication contraire lors de l'utilisation du symbole, la signification des symboles suivants est:

n	: nombre de symboles d'un mot de code.
k	: nombre de symboles d'un bloc d'information.
d_{min}	: distance minimale d'un code en bloc.
$\boldsymbol{d}$	: bloc d'information.
$\boldsymbol{c}$	: mot de code d'un code en bloc.
$wt(\boldsymbol{c})$	: poids de Hamming d'un mot de code $\boldsymbol{c}$.
$\boldsymbol{x}$	: mot de code modulé.
$\mathscr{A}$	: un alphabet d'éléments.
$\mathbb{F}_q$	: un corps de Galois à q éléments.
$\mathscr{C}(n,k,d_{min})$	: code en bloc de longueur n, de dimension k et de distance minimale d_{min}.
R	: rendement du code.
$\boldsymbol{G}$	: matrice génératrice d'un code en bloc.
$\boldsymbol{g}_i$	: la ième ligne de la matrice génératrice $\boldsymbol{G}$.
$\boldsymbol{H}$	: matrice de contrôle de parité d'un code en bloc.
$\boldsymbol{h}_i$	: la ième colonne de la matrice de contrôle $\boldsymbol{H}$.
$\boldsymbol{I}_k$	: matrice identitée de dimensions $k \times k$.
$\mathscr{C}^{\perp}$	: le code dual de $\mathscr{C}$.
$wt(\boldsymbol{c})$	: poids de Hamming d'un mot de code $\boldsymbol{c}$.
$\oplus$	: somme modulo 2.
$\otimes$	: produit cartésien des treillis.
$\langle \boldsymbol{g}_i \rangle$	: sous-code généré par le vecteur ligne $\boldsymbol{g}_i$.

N_0 : densité spectrale de puissance du bruit gaussien.

σ^2 : variance du bruit gaussien.

$\bar{\gamma}_t(s_t, s_{t+1})$: la métrique de branche reliant l'état s_t à l'état s_{t+1}.

$\Gamma_t(s)$: la métrique du chemin survivant à l'état s de l'étage t du treillis.

Δ_t^s : fiabilité du chemin survivant à l'état s de l'étage t du treillis.

$\boldsymbol{y}_{t_0}^{t_1}$: partie de $\boldsymbol{y}$ reçue entre les instants t_0 et t_1.

$\alpha_t(s)$: probabilité forward de l'état s de l'étage t du treillis.

$\beta_t(s)$: probabilité backward de l'état s de l'étage t du treillis.

$\gamma_t(s_t, s_{t+1})$: la probabilité de branche reliant l'état s_t à l'état s_{t+1}.

L_a : LLR *a priori*.

L_e : LLR extrinsèque.

it : itération.

π : permutation.

$\lceil a \rceil$: partie entière supérieure de a.

n_s^t : nombre d'états à l'étage t du treillis.

n_b^t : nombre de branches simples à l'étage t du treillis.

n_p : nombre des treillis-produits.

n_L : nombre de treillis élémentaires formant un treillis-produit.

n_S : nombre de sections groupées dans un treillis-produit.

m_t : nombre de bits dans un symbole dans la section t du treillis.

L_a^d : vecteur des LLRs *a priori* à l'entrée du décodeur d.

L_e^d : vecteur des LLRs extrinsèques délivré par le décodeur d.

N_b^t : nombre de branches simples composant une branche multiple.

N_d^t : nombre de branches multiples distinctes.

N_B^t : nombre total des branches multiples.

q_d^t : nombre d'occurrence de chaque branche multiple.

INTRODUCTION GÉNÉRALE

Ce mémoire résume mes 3 années de travail de recherche dans les Laboratoires Orange Labs de Recherche et Développement (R&D) de l'Opérateur de télécommunications Orange à Cesson-Sévigné près de Rennes (France). J'ai eu la chance d'y travailler pendant 3 années, de Décembre 2012 à Novembre 2015, sur un sujet à la fois bien théorique et bien pratique: **les codes correcteurs d'erreur en bloc courts et leur décodage à décision douce**.

Dans le cadre de cette thèse nous avons mis la barre très haut en termes d'objectifs de performances et de faible complexité d'implémentation des algorithmes de décodage. Mais comment ne pas mettre cette barre aussi haut pour contribuer en tant soit peu au niveau international à l'avancée de ce domaine des codes correcteurs d'erreur? Cette exigence de performances dans les résultats attendus est une nécessité pour passer le filtre des experts-lecteurs (reviewers) afin d'espérer avoir un article accepté à la publication dans une des meilleures conférences internationales. Cette contrainte de publication internationale nous impose aussi de mettre la barre très haut en termes d'originalité et de performances mais aussi de clarté d'exposé.

Ce domaine des codes correcteurs d'erreur est passionnant car à la fois très théorique et très pratique car motivé par un énorme marché industriel des télécommunications et des matériels électroniques: téléphones portables, ordinateurs, tablettes, disques durs, etc ... C'est toujours un marché en forte croissance tirée par un insatiable appétit des consommateurs pour toujours plus de débit et de services de télécommunications. Le nombre d'abonnements de téléphonie mobile se chiffre en Milliards et les générations 1G, 2G, 3G, ... de normes de téléphonie mobile se succèdent à un rythme effréné pour les capacités d'investissement des Opérateurs de télécommunications dits "telcos": 2G

en 1991, 3G en 2001, 4G en 2010, et bientôt la 5G prévue en 2020! Le graphique de la Figure 1 montre l'évolution des générations de normes de téléphonie mobile et leurs débits atteints ou envisagés:

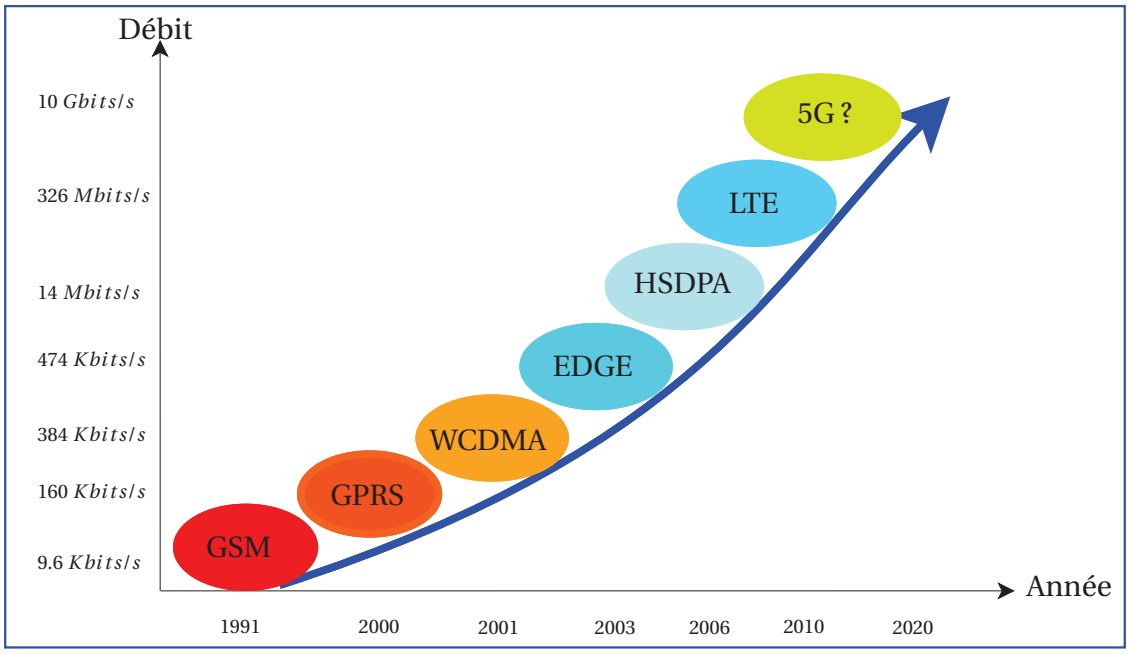

Figure 1. Evolution des technologies cellulaires d'aprs [1].

Les chiffres liés à Internet donnent le tournis. Et pourtant, aujourd'hui, seul 42% de la population mondiale est connectée à Internet. La création de données numériques n'a jamais été aussi féconde et l'augmentation est exponentielle. Le nombre d'utilisateurs de l'Internet est passé d'environ 1.5 milliards en 2008 à à peu près 3 milliards d'utilisateurs en 2014. En France, le nombre d'internautes devrait dépasser les 50 millions en 2016 et atteindre 52.5 millions en 2018.

La Figure 2 présente l'évolution du nombre d'utilisateurs d'Internet dans le monde entier entre 2008 et 2014.

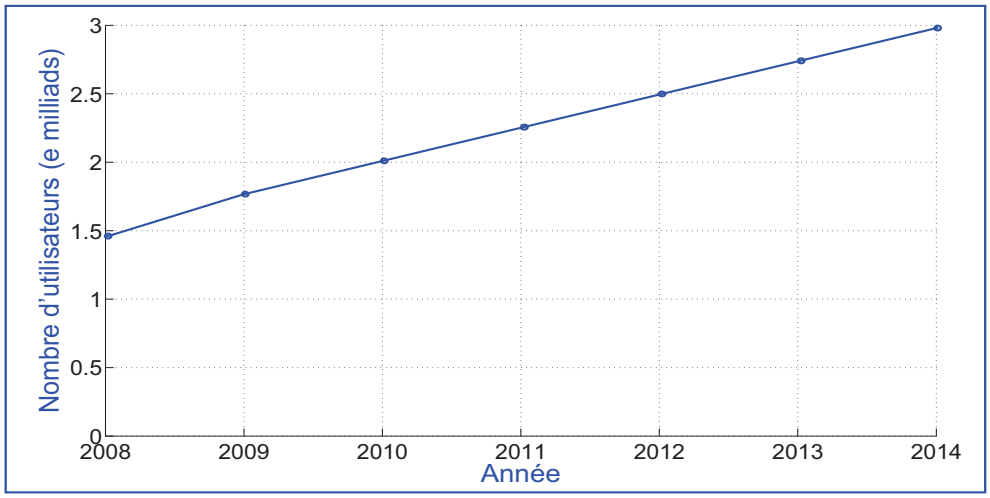

Figure 2. Nombre d'utilisateurs d'Internet entre 2008 et 2014 [2].

Le nombre d'objets connectés à internet est estimé en cette année 2015 à plus de 18 milliards d'objets [3] utilisés dans plusieurs domaines: télé-santé, domotique, capteurs d'activités personnelles (quantified self), les systèmes électriques intelligents (smart grid), la détection environnementale (capteurs de température, capteurs d'humidité,...etc),...etc et il est prévu que ce nombre passera d'ici 2020 à plus de 50 milliards!

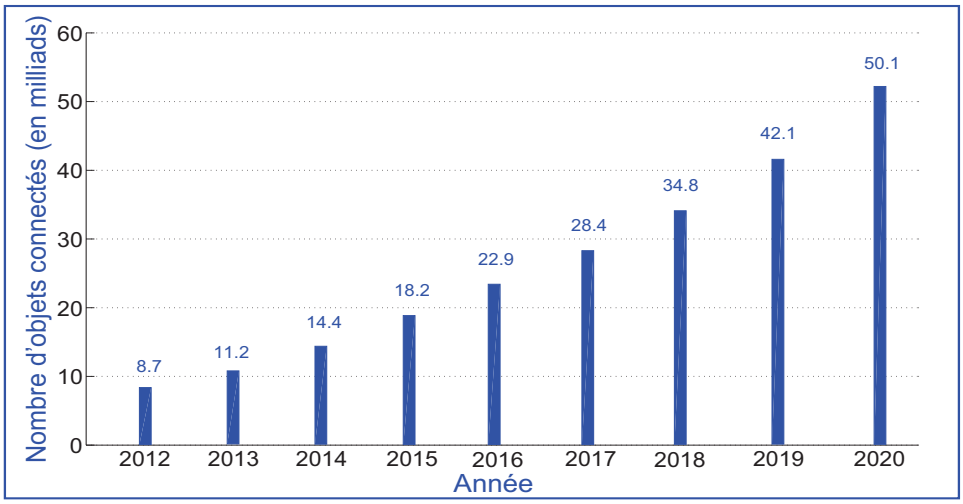

Figure 3. Nombre d'objets connects prvus d'ici 2020 [3].

Ces nombreux services exigent une grande qualité et fiabilité dans la transmission des données. Les systèmes de correction d'erreur y contribuent de façon très peu apparente à l'utilisateur final mais de façon si indispensable et efficace au niveau le plus basique de la transmission: la couche physique. Le travail de recherche dans la couche physique nous apprend donc la modestie et l'humilité car les codes correcteurs sont absolument omniprésents et indispensables aux télécommunications mais jamais mis en lumière. Nous y sommes donc à l'abri du tapage médiatique et des modes. C'est pourtant un domaine humainement enrichissant car tous ces petits téléphones portables recèlent tous des petits bouts des mêmes logiciels que la Communauté des Télécommunications a pensés, discutés, écrits, testés!

Un système de transmission fiable est donc un système qui permet de véhiculer des données d'un point à un autre avec le moins d'erreurs possible tout en maximisant le débit de transmission. Cette exigence de qualité finalement assez facilement quantifiable a donné naissance en 1948 à la théorie de l'information débutant par l'article fondateur de Claude Shannon[4]. Cet article de Shannon démontre qu'un tel système peut être réalisé en utilisant un codage convenable de l'information à condition que le débit de transmission envisagé ne dépasse pas la limite de la capacité du canal. Il faut ici rendre hommage à Richard Hamming qui a inventé les premiers codes correcteurs non-triviaux en 1947. Claude Shannon a écrit ses articles fondateurs en connaissant bien les travaux de Hamming sur ses codes correcteurs. Claude Shannon et Richard Hamming occupèrent un temps le même bureau aux Bell Labs! D'après Hamming, la hiérarchie des Bell Labs les considérait tous deux comme de si insupportables joyeux drilles qu'ils furent mis ensemble à l'écart dans ce "placard" pendant un certain temps! Bien que cela date de bientôt 70 ans, les travaux de tous nos brillants Prédécesseurs nous sont très proches par leurs articles consultables par Internet dans la base de donnée d'articles "Xplore" de l'IEEE[1] et sont intemporels. Nous célébrerons en 2016 le 100ème anniver-

[1] Institute of Electrical and Electronics Engineers

saire de la naissance de Claude Shannon.

L'histoire de l'invention des codes de Hamming mérite bien quelques lignes. Richard Hamming travaillait en 1947 aux Bells labs comme scientifique et programmeur d'un des tout premier calculateur. Un vendredi avant de partir en week-end, il "lance" un long programme de calcul en espérant récupérer la solution du programme complexe le lundi suivant. Mais de retour au travail ce lundi, le programme avait été arrêté à cause d'une erreur matérielle dans les circuits logiques à relais. Il entrepris de résoudre lui-même le problème et de proposer une solution matérielle. Comme les calculs étaient effectués en BCD (Binaire Codé Décimal): 4 bits codaient les entiers de 0 à 9, il ajouta 3 bits de redondance pour faire un mot de 7 bits dans lesquels on pouvait corriger une erreur et en détecter deux! Les codes de Hamming étaient nés. Richard Hamming a publié en 1950 ses codes en bloc linéaires capables de corriger une seule erreur [5].

Deux des plus prestigieux prix scientifiques créés par l'IEEE et décernés tous les ans sont l'IEEE Shannon Award [6] et la Gold Hamming Medal [7]. Claude Berrou, Alain Glavieux et Thitimajshima ont reçu la Gold Hamming Medal en 2003 pour l'invention en 1991 et publication en 1993 des "**turbocodes**" qui ont révolutionné les télécommunications. Les travaux des lauréats de l'IEEE Shannon Award sont aussi à l'origine des avancées majeures du domaine des télécommunications avec comme exemples: Gallager en 1983, Viterbi en 1991, Forney en 1995, Sloane en 1998, Kasami en 1999, Wolf en 2001, McEliece en 2004, et Calderbank en 2015.

Depuis 1948, c'est donc une floraison ininterrompue d'inventions des codes correcteurs d'erreur.

En 1954, Müller invente une nouvelle classe de codes pour la détection d'erreurs [8] pour lesquels Reed propose ensuite un algorithme de décodage pour la correction d'erreurs [9]. Ces codes sont connus sous le nom de "Reed-Müller"(RM).

En 1955, Elias invente les codes convolutifs [10]. Il invente aussi presque en même temps les codes produits [11] que Pyndiah réussit à décoder de façon itérative quasi-optimale et peu complexe en 1995 [12].

En 1957 Prange invente les codes cycliques [13]. Deux ans plus tard et en 1959 Bose, Chaudhury et Hockenghem inventent une nouvelle classe des codes cycliques, appelés BCH (acronyme des initiales des noms des 3 inventeurs) [14][15] .

En 1960 Reed et Solomon inventent les codes Reed-Solomon [16] qui sont la classe des codes cycliques la plus utilisée en pratique.

En 1961 Gallager invente, dans le cadre de sa thèse, les codes LDPC (Low-Density Parity-Check) [17].

En 1966 Forney invente les codes concaténés [18]. Il propose un schéma de codage en concaténant en série deux codes en bloc: un code de Reed-Solomon et un code binaire.

Après les années 1960, les progrès dans le domaine des codes correcteurs d'erreur se faisant apparemment rares, surgit une polémique "Coding is dead?" au premier congrès de "IEEE Communication Theory Workshop" qui se tenait en Floride en avril 1971 [19].

Il a fallu attendre l'année 1993 pour avoir une réponse à cette question pessimiste. Claude Berrou, Alain Glavieux et Thitimajshima [20] présentent, à la conférence ICC'93, les turbocodes permettant d'atteindre la limite de Shannon à quelques fractions de dB près. L'équipe de Claude Berrou de l'école Télécom Bretagne a présenté à cette même conférence ICC'93 une puce VLSI réalisant le décodage des turbocodes en temps réel rendant obsolète le coûteux projet à 1 million de dollars du "Big-Viterbi Decoder" [21] à $2^{16} = 65536$ états! Les turbocodes ont vraiment été une révolution pour le domaine des télécommunications. C'est d'ailleurs le terme exact de "révolution" donné par l'IEEE avec l'attribution de la Gold Hamming Medal aux Inventeurs.

L'invention des turbocodes a ensuite suscité la redécouverte des codes LDPC par MacKay en 1995 [22] après avoir été oublié pendant plus de 30 ans. Cet oubli prolongé s'explique par la trop grande complexité des calculs de décodage des codes LDPC pour les technologies disponibles en 1961. L'explosion de la puissance de calcul des circuits intégrés sur silicium à grande échelle ou VLSI (Very Large Scale Integrated) la rendant possible sur une seule puce 1991. Cette croissance exponentielle, depuis les années 1970, de la capacité d'intégration sur silicium est bien connue sous le nom de "Loi de Moore [23]" du nom de Gordon Moore l'un des fondateurs de la Société Intel.

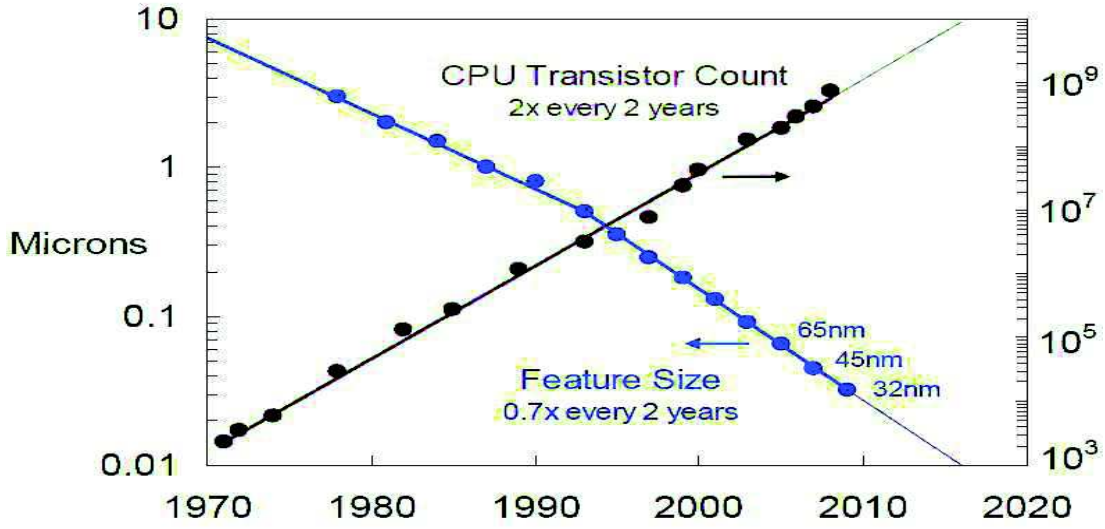

Figure 4. Evolution des dimensions minimales de gravure en microns(μm) des circuits VLSI et de leurs densits en nombre de transistors/mm^2 compars la loi de Moore entre 1970 et 2020 d'aprs [24].

Les techniques de décodage aussi ont évolué en parallèle de l'invention de ces familles de codes de plus en plus complexes et puissantes pour corriger des erreurs.

Nous pouvons considérer que le début de l'emploi des techniques de décodage à "décision douce", c'est-à-dire utilisant toute l'information d'une variable reçue quantifiée sur de multiples niveaux 4, 8, 16, 32, ...et non plus seulement avec son signe 0 ou 1 encore appelée "décision dure", remonte au décodage majoritaire employé pour la

première fois pour le décodage des codes de Reed-Müller(RM) en 1954 [9]. Le décodage majoritaire consiste à ajouter les répliques d'une même variable binaire et de décider 0 ou 1 s'il y a une majorité de répliques à 0 ou 1 parmi l'ensemble de ces répliques d'une même variable. Le terme "réplique" provient des travaux de Gérard Battail [25–27] sur le décodage à décision douce dans les années 1970.

Le premier algorithme introduit pour le décodage des codes convolutifs a été inventé par Wozencraft en 1957 [28]. Cet algorithme, appelé décodage séquentiel, est ensuite amélioré par Fano [29] en 1963.

En 1961, Gallager invente, en même temps que les codes LDPC, le fameux algorithme de décodage Belief Propagation (BP)[17].

L'algorithme de décodage qui fait date est l'algorithme de Viterbi proposé en 1967 par Viterbi [30] comme méthode de décodage pour les codes convolutifs. En 1969, Omura montre dans [31] que l'algorithme de Viterbi est similaire à une phase en-avant ou forward d'un algorithme de programmation dynamique proposé par Bellman en 1957 [32].

En 1973 Forney [33] montre tout l'intérêt de cet algorithme de Viterbi pour faire en temps réel le décodage des codes convolutifs mais plus généralement de tous les codes décrits par un treillis. Forney présente en même temps, la notion d'un treillis pour les codes convolutifs.

En 1974 Bahl, Cocke, Jelinek et Raviv présentent un nouvel algorithme [34] qui permet d'estimer de façon optimale les probabilités *a posteriori* par bit à partir des probabilités *a priori* par bit reçu et du treillis du code. Ils présentent en même temps et pour la première fois une méthode simple pour calculer un treillis d'un code en bloc à partir de sa matrice de contrôle.

En 1978, Wolf [35] présente une méthode pour construire un treillis d'un code en bloc similaire à la méthode BCJR et détermine une borne supérieure pour le nombre d'états de ce treillis. Cette publication était la première étude approfondie et dédiée à la structure en treillis des codes en bloc.

En cette même année 1978, Massey publie un papier [36] dans lequel il présente une nouvelle méthode pour calculer un treillis d'un code en bloc et donne une définition plus précise d'un treillis d'un code et quelques propriétés fondamentales de la structure du treillis.

Cependant, ces premiers travaux sur les treillis des codes en bloc n'ont pas attiré beaucoup d'attention et les recherches dans ce domaine resteront inactives pendant les 10 prochaines années. Cette période d'inactivité dans ce domaine est justifiée par le fait que la majorité des chercheurs à cette époque croyaient fortement que les codes en bloc ne pouvaient pas avoir une structure en treillis simple comme les codes convolutifs et donc l'impossibilité de leur décodage par l'algorithme de Viterbi (à l'exception de quelques codes courts) et que la seule manière de décoder les codes en bloc était d'utiliser le décodage algébrique.

Il a fallu attendre 10 ans avant que Forney publie en 1988 son papier [37] qui a remis ce domaine au goût du jour. Dans ce papier Forney montre que certains codes en bloc comme les codes Reed-Müller (RM) peuvent avoir des treillis simples et il y présente une méthode pour calculer un treillis minimal d'un code c'est-à-dire le treillis qui a le plus petit nombre d'états de tous les treillis décrivant le code.

Depuis cette date, les recherches dans le domaine des treillis des codes en bloc se sont intensifiées conduisant ensuite au développement rapide du domaine et surtout à la conception des algorithmes de décodage à décision douce et de bonnes performances pour les codes en bloc [38–44].

Les représentations en treillis des codes conduisent à la conception des algorithmes de décodage de performances optimales ou quasi-optimales avec une complexité réduite [45]. Cette réalité a interféré avec nos objectifs, en termes de conception d'algorithmes de décodage, en nous a encourageant beaucoup à aller dans la direction de décodage des codes en bloc sur treillis

En 1989, Hagenauer présente une variante de l'algorithme de Viterbi appelée SOVA (Soft-Output Viterbi Algorithm) [46] permettant d'associer des fiabilités aux décisions dures de l'algorithme de Viterbi.

En 1990, Koch *et al* propose une variante de l'algorithme BCJR moins complexe et moins performante appelée Max-Log-MAP [47]. Cinq ans plus tard et en 1995, Robertson *et al* présente une autre variante de l'algorithme BCJR appelée Log-MAP [48] qui permet de réduire sa complexité sans dégrader ses performances.

En 1996, Wiberg montre, dans le cadre de sa thèse [49], que le décodage des turbocodes et les codes LDPC peut se faire sur graphes creux (en anglais sparse graph) en utilisant l'algorithme de décodage MS (Min-Sum Algorithm). Wiberg re-découvre les travaux de Tanner et ses graphes [50] créant ensuite un nouveau domaine appelé "codes sur graphe" ou bien en anglais "codes on graph".

La publication des turbocodes en 1993 à ICC'93 la Conférence Internationale des Communications en Mai 1993 [20] a provoqué une floraison d'études, de publications et d'inventions comparables à celle de 1951 après l'invention des codes de Hamming.

En ces années 1990, dans la foulée des turbocodes, de multiples variantes de familles de codes concaténés sont inventées dont les codes Cortex inventés en 1998 par Carlach et Vervoux [51]. Les codes Cortex constituent une famille de codes en bloc de paramètres $(n = 2 * k, k, d_{min})$ construits avec de petits codes de base concaténés en série et en parallèle où n, k et d_{min} sont respectivement la longueur, la dimension et la distance minimale du code. Nous reprendrons en détails cette famille des codes dans le deuxième chapitre de ce manuscrit de thèse.

Dans la suite nous citons les dates de quelques inventions "récentes" qui, entre autres, représentent une avancée dans le domaine des codes correcteurs d'erreurs.

En 1999, Ten Brink invente la notion de "EXIT Chart" ou courbe de transfert d'information extrinsèque pour étudier la convergence des décodages itératifs [52][53].

En 2001, Berrou *et al.* frappe encore un coup en perfectionnant les turbocodes par l'invention des turbocodes duobinaires [54].

En 2009 Arikan [55] reprend une structure d'encodage ressemblant à celle des codes Cortex pour créer les "Polar Codes" pour des codes en bloc de grandes longueurs et surtout proposer des algorithmes de décodage efficaces sur ces structures.

Les années 2000 voient aussi un nouveau domaine émergé, les codes correcteurs quantiques [56]. Les codes correcteurs quantique ou QEC (Quantum Error Correction) permettent de combattre la fragilité des états quantiques qui empêche de compléter les algorithmes quantiques et donc l'accélération de développement des ordinateurs quantiques.

Depuis le début de ma thèse en 2012 et encore maintenant en 2015, peut-on s'aventurer à poser la question: "Coding is Dead"? Les turbocodes et les codes LDPC sont bien installés et reconnus dans les normes jusqu'à la 4G. N'y-a-t-il donc plus rien à découvrir ou inventer? Il suffit d'ouvrir une des revues mensuelles de l'IEEE comme IEEE Transactions on Information Theory, IEEE Transactions on Communications ou IEEE Transactions on Wireless Communications, ... pour se rendre compte de l'ineptie d'une telle assertion. Jamais la théorie de l'information et ses domaines connexes comme les codes correcteurs d'erreur n'ont été aussi florissants!

Cependant, quelques "petits" problèmes restent non-résolus. En effet les turbocodes et les codes LDPC ou leurs variantes ont d'excellentes performances pour des longueurs n supérieures ou égales de l'ordre de quelques milliers de bits ($n > 1000$) et pour une complexité raisonnable des implémentations en temps réel même pour des débits très élevés (GigaBit/s).

Par contre pour les petites longueurs ($n < 1000$) de codes, et c'est très paradoxal, le décodage de turbocodes et codes LDPC courts est moins performant. En effet, pour avoir de grande distances minimales d_{min} avec des codes de petites longueurs n et de grandes capacités de correction, il faut des codes dont les matrices de contrôle de parité soient de densité élevée et non creuse, c'est à dire avec beaucoup de 0 et très peu de 1, comme celles des codes LDPC ou des turbocodes. Or, comme une matrice dense implique des cycles courts dans le graphe de Tanner du code, l'algorithme Belief-

Propagation (BP) de Gallager sur ce graphe de Tanner est très inefficace à cause des cycles de longueur très courtes qui ramènent, la même information extrinsèque à l'endroit où elle a été produite!

L'objectif principal de cette thèse est donc l'étude des algorithmes de décodage à décision douce pour les codes en bloc courts.

Nous nous sommes intéressés dans un premier temps à l'étude d'une classe particulière de codes en bloc dits Cortex [51] dont le décodage généralisé est encore non-résolu sauf pour des cas très particuliers comme le code de Golay$(24,12,8)$. L'objectif très ambitieux fixé dans ce cadre est de mettre en oeuvre des algorithmes de décodage qui tirent avantage de la structure de codage factorisée des codes Cortex mais sans être contraints par la longueur du code. Cependant, toutes nos tentatives de décodage ont été confrontées au problème de variables cachées dans cette structure et à la difficulté de leurs estimations à partir des symboles reçus.

Dans un deuxième temps, nous avons élargi nos recherches en fixant de nouveaux objectifs, encore très ambitieux, de proposer des algorithmes de décodage très généraux valables pour tout code en bloc. Nous nous sommes posé tout d'abord la question sur le point de départ qui doit être un point commun entre tous les codes en bloc: la représentation la plus commune entre codes en bloc est la représentation matricielle soit par la matrice génératrice ou par la matrice du contrôle du code. Avec ces objectifs ambitieux, nous avions un oeil ouvert sur les travaux déjà effectués sur les treillis des codes en bloc et les opportunités que ces treillis offrent pour concevoir des algorithmes de décodage de bonnes performances. Nous devons citer dans cadre les travaux de F. Kschischang *et al* [57] et Calderbank *et al* [41] sur les treillis des codes en bloc. Dans ces travaux, est proposée une méthode permettant de calculer un treillis global d'un code en bloc par le produit cartésien des plusieurs treillis élémentaires à 2 états issus des lignes de la matrice génératrice ou la matrice de contrôle du code. Un nouvel axe de recherche est ensuite né. Il s'agit d'étudier les possibilités de mettre en oeuvre des schémas de décodage itératif basés sur des treillis de complexité réduite construits à partir de ces

treillis élémentaires à 2 états. Les résultats de cet axe de recherche sont présentés dans les deux chapitres 3 et 4.

Ce manuscrit de thèse est organisé en quatre chapitres.

Le premier chapitre résume brièvement l'état de l'art de codes correcteurs d'erreurs. Il présente les codes en bloc. Il donne ensuite une description de la notion d'un treillis, qui est fondamentale dans nos travaux, et présente quelques méthodes permettant de le calculer pour un code en bloc. Il présente également les algorithmes de décodage les plus utilisés resp. l'algorithme de Viterbi et l'algorithme BCJR, ainsi que leurs variantes resp. SOVA (Soft Viterbi Algorithm) et Max-Log-Map. A la fin, il présente la notion du décodage itératif et l'explicite à l'aide d'un exemple d'un turbocode.

Le deuxième chapitre est consacré à l'étude des codes Cortex. Il donne tout d'abord un bref résumé des plus intéressants travaux de recherches effectués autour des codes Cortex. Il présente ensuite la structure d'encodage des codes Cortex et donne quelques exemples de constructions. Finalement, il aborde le sujet de décodage en présentant les difficultés encore posées devant la conception des algorithmes de décodage basés sur la structure. Il présente dans ce cadre certains décodages qui ont été effectués puis les résultats de nos travaux de recherche même si ce sont des échecs qui nous ont beaucoup appris.

Le troisième chapitre, présente un algorithme original de décodage des codes en bloc. Cet algorithme fait interagir des treillis élémentaires issus de la matrice génératrice (ou de contrôle) du code 2 à 2 sur plusieurs étages afin de calculer des estimations des fiabilités ou décisions douces encore appelées Log-Likehood Ratio (LLR) *a posteriori* sur les symboles reçus. Il présente la notion d'un treillis élémentaire, définit le produit cartésien et expose la méthode à l'aide d'un code de Hamming(8,4,4). Les performances de l'algorithme sont ensuite abordées avec une étude détaillée de sa complexité.

Le quatrième chapitre, présente deux algorithmes originaux de décodage itératif pour les codes en bloc basés sur des treillis-produits. Ces deux algorithmes utilisent des treillis-produits de complexité réduite construits à partir des treillis élémentaires représentant des lignes de la matrice de contrôle du code. Le premier algorithme fait une recherche du mot le plus probable sur chacun des treillis-produits au lieu de faire la recherche sur le treillis global du code. Le deuxième algorithme est un algorithme à décision douce en entrée et en sortie (SISO: Soft-In Soft-Out) qui délivre en sortie une estimation des LLRs *a posteriori* des symboles reçus.

Le cinquième chapitre est une conclusion où nous rappelons les résultats obtenus et donnons les perspectives aux travaux présentés dans cette thèse.

Chapitre 1

NOTIONS DE BASE SUR LES CODES CORRECTEURS D'ERREUR

1.1 Introduction

Le but premier des codes correcteurs d'erreur est d'augmenter la fiabilité des communications en présence de bruits et d'interférences en protégeant au maximum l'information utile des perturbations. Le canal de transmission peut être perturbé par du bruit, au point de détruire l'information émise. Le codage est l'opération faite sur l'information avant sa transmission. Il consiste en l'insertion d'une quantité de redondance avec les données utiles à envoyer afin de nous permettre de restituer l'information émise à la réception. Cette opération de restitution à la réception s'appelle le décodage. La Figure 1.1 présente un modèle d'une chaine de communications numériques.

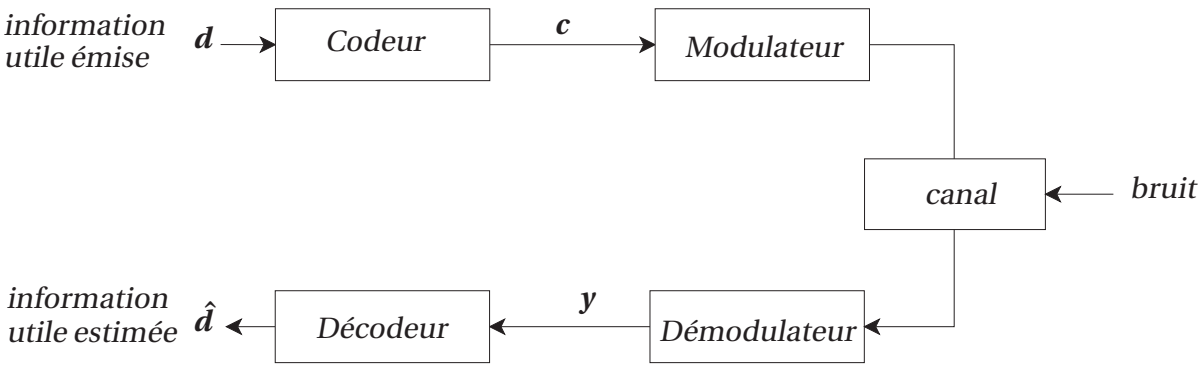

Figure 1.1. Modle d'une chaine de communication

1.2 Les codes en bloc linéaires

Le codage en bloc consiste à associer à un bloc d'information $\boldsymbol{d}$ de k symboles un bloc $\boldsymbol{c}$, appelé mot de code, de n symboles avec $n \geq k$. Le code introduit donc une quantité

de redondance de $(n-k)$ symboles. Les symboles prennent leurs valeurs dans un alphabet $\mathscr{A}$. Les alphabets les plus employés en pratique sont les corps de Galois $\mathbb{F}_q$ à q éléments [59].

Un code en bloc est une application f de l'ensemble $\mathbb{F}_q^k$ vers l'ensemble $\mathbb{F}_q^n$ qui associe à un bloc d'information $\boldsymbol{x}$ un mot de code $\boldsymbol{c}$:

$$f \left| \begin{array}{ccc} \mathbb{F}_q^k & \rightsquigarrow & \mathbb{F}_q^n \\ \boldsymbol{d} & \longmapsto & \boldsymbol{c} = f(\boldsymbol{x}) \end{array} \right.$$

Les paramètres k et n sont appelés respectivement la dimension et la longueur du code en bloc que nous notons $\mathscr{C}(n,k)$. Le rendement du code $\mathscr{C}$, noté R, est un rapport entre la dimension k et la longueur n du code: $R = k/n$.

Si $q = 2$, les symboles prennent leurs valeurs dans $\mathbb{F}_2 = \{0,1\}$ et le code est dit binaire. Dorénavant, nous allons considérer seulement les codes en bloc binaires.

1.2.1 Linéarité

Un code en bloc $\mathscr{C}(n,k)$, défini sur $\mathbb{F}_2$ est dit linéaire si et seulement si ses 2^k mots de code constituent un sous-espace vectoriel de dimension k de l'espace vectoriel $\mathbb{F}_2^n$ c'est-à-dire si l'application f est linéaire. La linéarité implique que la somme de deux mots de code est aussi un mot de code et que le mot tout à zéro est un mot de code.

1.2.2 Matrice génératrice

Un code en bloc linéaire $\mathscr{C}(n,k)$, défini sur $\mathbb{F}_2$, associe à un bloc d'information $\boldsymbol{d} = (d_0, d_1, \ldots, d_{k-1})$ de k bits, un bloc $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ de n bits. Soient $(\boldsymbol{b}_0, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{k-1})$ et $(\boldsymbol{b}'_0, \boldsymbol{b}'_1, \ldots, \boldsymbol{b}'_{n-1})$ deux bases des ensembles $\mathbb{F}_2^k$ et $\mathbb{F}_2^n$ respectivement. Donc le vecteur $\boldsymbol{d}$ peut être décrit par:

$$\boldsymbol{d} = \sum_{i=0}^{k-1} d_i.\boldsymbol{b}_i \tag{1.1}$$

Comme l'application f est linéaire, le mot $\boldsymbol{c}$ est donné par:

$$\boldsymbol{c} = f(\boldsymbol{d}) = \sum_{i=0}^{k-1} d_i . f(\boldsymbol{b}_i) \tag{1.2}$$

Comme $f(\boldsymbol{b}_i) \in \mathbb{F}_2^n$, alors ils existent n valeurs binaires $g_{i0}, g_{i1}, \ldots, g_{i(n-1)}$ telles que:

$$f(\boldsymbol{b}_i) = \sum_{m=0}^{n-1} g_{im} . \boldsymbol{b}'_i \tag{1.3}$$

Soit $\boldsymbol{G}$ la matrice constituées des lignes $\boldsymbol{g}_i = f(\boldsymbol{b}_i)$, $i = 0, 1, \ldots, k-1$:

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{g_0} \\ \boldsymbol{g_1} \\ \vdots \\ \boldsymbol{g_{k-1}} \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0(n-1)} \\ g_{10} & g_{11} & \cdots & g_{1(n-1)} \\ \vdots & \vdots & \vdots & \\ g_{(k-1)0} & g_{(k-1)1} & \cdots & g_{(k-1)(n-1)} \end{bmatrix} \tag{1.4}$$

Le mot $\boldsymbol{c}$ peut donc être exprimé sous forme d'un produit matriciel entre le vecteur $\boldsymbol{x}$ et la matrice $\boldsymbol{G}$ tel que:

$$\boldsymbol{c} = \boldsymbol{d} \cdot \boldsymbol{G} \tag{1.5}$$

La matrice $\boldsymbol{G}$ de dimensions $k \times n$ est appelée matrice génératrice du code $\mathscr{C}$.

Le code en bloc $\mathscr{C}$ n'a pas une seule matrice génératrice. En effet, toute permutation des vecteurs de la base $(\boldsymbol{b}_0, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_{k-1})$ (resp. de la base $(\boldsymbol{b}'_0, \boldsymbol{b}'_1, \ldots, \boldsymbol{b}'_{n-1})$) conduit à une base de $\mathbb{F}_2^k$ (resp. de $\mathbb{F}_2^n$) et donc à une nouvelle matrice génératrice du code. En d'autre termes, toute permutation des lignes ou de colonnes de la matrice $\boldsymbol{G}$ conduit à une autre matrice génératrice du code $\mathscr{C}$.

1.2.3 Matrice de contrôle de parité

Une matrice de contrôle de parité $\boldsymbol{H}$ du code $\mathscr{C}$ vérifie, pour tout mot de code $\boldsymbol{c}$ de $\mathscr{C}$, la relation suivante :

$$\boldsymbol{c} \cdot \boldsymbol{H}^T = 0 \tag{1.6}$$

où $\boldsymbol{H}^T$ désigne la matrice transposée de $\boldsymbol{H}$.

La matrice $\boldsymbol{H}$ permet au décodeur de détecter la présence d'erreurs sur le mot reçu car

il suffit que son produit avec $\boldsymbol{H}^T$ soit différent de $\mathbf{0}$. C'est pour cette raison qu'elle est appelée matrice de contrôle de parité.

Si la matrice $\boldsymbol{G}$ est sous forme systématique c'est-à-dire qu'elle s'écrit sous la forme :

$$\boldsymbol{G} = [\boldsymbol{I}_k | \boldsymbol{P}] \tag{1.7}$$

où $\boldsymbol{I}_k$ est une matrice d'identité de dimensions $k \times k$. Alors la matrice de contrôle $\boldsymbol{H}$ est donnée par:

$$\boldsymbol{H} = [\boldsymbol{P}^T | \boldsymbol{I}_{n-k}] \tag{1.8}$$

où $\boldsymbol{P}^T$ désigne la matrice transposée de $\boldsymbol{P}$.

Nous pouvons associer au code en bloc linéaire $\mathscr{C}$, un code en bloc linéaire qu'on note $\mathscr{C}^\perp$ de dimension $(n-k)$ et de longueur n, qui vérifie que tout mot du code $\mathscr{C}^\perp$ est orthogonal à tout mot du code $\mathscr{C}$ c'est-à-dire le produit scalaire entre un mot de $\mathscr{C}$ et un mot de $\mathscr{C}^\perp$ est nul. Le code $\mathscr{C}^\perp$ est appelé code dual de $\mathscr{C}$.

La matrice de contrôle $\boldsymbol{H}$ du code $\mathscr{C}$ est une matrice génératrice pour $\mathscr{C}^\perp$.

1.2.4 Distance minimale

Le poids de Hamming, noté $wt(\boldsymbol{c})$, d'un mot de code $\boldsymbol{c}$ d'un code en bloc $\mathscr{C}$ est le nombre de ses éléments non nuls. Le poids minimal, noté w_{min}, d'un code $\mathscr{C}$ est le plus petit poids de Hamming de tous les mots de code non nuls.

$$w_{min} = \min_{\boldsymbol{c} \in \mathscr{C}, \boldsymbol{c} \neq \mathbf{0}} wt(\boldsymbol{c}) \tag{1.9}$$

Pour un code linéaire, sa distance minimale, notée d_{min}, est égale à son poids minimal.

$$d_{min} = w_{min} \tag{1.10}$$

Un code linéaire $\mathscr{C}$ de longueur n, de dimension k et de distance minimale d_{min} est noté $\mathscr{C}(n, k, d_{min})$.

1.2.5 Capacité de détection et de correction d'erreur

Pour un code en bloc $\mathscr{C}$ de distance minimale d_{min}, deux mots de code se distinguent sur au moins d_{min} positions. Cela signifie que tout motif (ou pattern) de t erreurs ($t \leq (d_{min}-1)$), dans un mot de code, ne peut pas le transformer en un autre mot de code. Nous pouvons donc détecter jusqu'à $d_{min}-1$ erreurs dans un mot de code. Cependant, nous ne pouvons pas détecter tous les motifs de d_{min} erreurs. En effet, il existe au moins une paire de mots de code qui se distinguent exactement en d_{min} positions et donc un motif de d_{min} erreurs peut transformer l'un de ces deux mots en l'autre. Le même argument s'applique aux patterns de plus de d_{min} erreurs. Pour cette raison, nous disons que la capacité de détection d'erreurs d'un code en bloc de distance minimale d_{min} est $d_{min}-1$. Parmi les erreurs detectées, le code $\mathscr{C}$ peut garantir la correction d'au moins t erreurs tel que:

$$t = \lfloor (d_{min}-1)/2 \rfloor \tag{1.11}$$

où $\lfloor . \rfloor$ est l'opérateur partie entière inférieure.

1.2.6 Les représentations en treillis des codes en bloc

Dans ce paragraphe, nous allons tout d'abord donner une définition formelle d'un treillis puis nous allons l'exposer à l'aide d'un exemple d'un petit code en bloc binaire $(4,2)$. Le calcul d'un treillis d'un code en bloc est ensuite abordé en présentant dans les sections 1.2.6.1 et 1.2.6.2 deux méthodes permettant de le calculer. La section 1.2.6.3 étudie la complexité d'un treillis en l'évaluant en termes du nombre d'états et de branches.

Soit $\mathscr{C}(n,k,d_{min})$ un code en bloc linéaire défini sur $\mathbb{F}_2$. Le treillis décrivant le code en bloc $\mathscr{C}$ est un graphe noté $T = (V,E,\mathbb{F}_2)$ de longueur n où l'ensemble V est partitionné à $n+1$ étages d'états telles que:

$$V = V_0 \cup V_1 \cup \ldots \cup V_n, \quad \text{avec} \quad |V_0| = |V_n| \tag{1.12}$$

Et E est un ensemble des branches telles que chaque branche dans E est étiquetée d'un bit de $\mathbb{F}_2$ et elle commence d'un état de V_t et se termine en un autre état de V_{t+1} pour

$t \in \{0,1,\ldots,n-1\}$. L'ensemble d'indices $\{0,1,\ldots,n-1\}$ introduit par la partition dans l'équation 1.12 est appelé l'axe du temps du treillis T.

Dans ce treillis T, un chemin est composé d'une succession de branches allant d'un état de l'étage V_0 à un état de l'étage V_n. Il y a une correspondance bijective entre l'ensemble des chemins de T et l'ensemble des mots du code $\mathscr{C}$.

Dans le cas où le treillis commence par un seul état (par convention l'état 0) et se termine en seul état (c'est-à-dire $|V_0| = |V_n| = 1$), il est dit "classique". Dans le cas contraire où $|V_0| = |V_n| > 1$, le treillis est dit "tail-biting" [41–44].

Exemple: Treillis décrivant le code en bloc de paramètres $(n = 4, k = 2)$

Un treillis décrivant le code $(4,2)$ est représenté sur la Figure 1.2. Dans ce treillis, il y a une correspondance bijective entre l'ensemble de ses chemins et l'ensemble des mots du code $(4,2)$ composé de 4 mots suivants: $\{(0,0,0,0), (0,1,1,0), (1,0,1,1), (1,1,0,1)\}$. Le calcul d'un treillis d'un code en bloc, et en particulier le calcul de ce treillis, est abordé dans les paragraphes suivants.

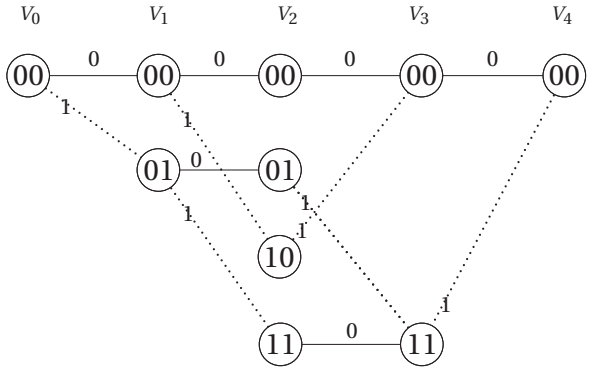

Figure 1.2. Treillis dcrivant le code binaire $(4,2)$.

Remarque: Dans tout ce rapport, une "section" d'un treillis signifie la partie composée de deux étages d'états reliés par un ensemble de branches. La section t du treillis T est composée de deux étages d'états V_{t-1} et V_t et l'ensemble des branches de E qui les relie. Donc dans le treillis T, il y a $(n+1)$ étages et n sections.

Dans la suite, nous allons présenter deux méthodes permettant de calculer un treillis

d'un code en bloc. Ces deux méthodes seront utilisées au cours de ce rapport.

1.2.6.1 Construction Wolf-BCJR

Cette méthode a été proposée en 1974 par Bahl, Cocke, Jelinek et Raviv [34], en même temps que l'algorithme de décodage BCJR, pour calculer un treillis d'un code en bloc à partir de sa matrice de contrôle de parité. Les travaux ultérieurs de Wolf [35] en 1978 présentent une méthode similaire et montrent que le nombre d'états "actifs" à un étage t du treillis est au plus $2^{(n-k)}$ où k et n sont la dimension et la longueur du code en bloc.

Dans ce paragraphe, nous allons tout d'abord donner une description des étapes effectuées par la méthode pour calculer un treillis d'un code en bloc linéaire. Puis nous l'explicitons à l'aide d'un exemple simple du code en bloc binaire $(4,2)$. Pour décrire cette méthode nous considérons un code en bloc linéaire $\mathscr{C}(n,k)$, défini sur $\mathbb{F}_2$ et dont une matrice de contrôle est notée $\boldsymbol{H} = [\boldsymbol{h}_0, \boldsymbol{h}_1, \ldots, \boldsymbol{h}_{n-1}]$, telle que $\boldsymbol{h}_i \in \mathbb{F}_2^{(n-k)}$, $0 \leqslant i \leqslant n-1$, sont les colonnes de $\boldsymbol{H}$.

Les étapes effectuées par la méthode BCJR pour calculer le treillis $T = (V, E, \mathbb{F}_2)$ du code $\mathscr{C}$ sont décrites telles que:

1. Le treillis commence sur l'état zéro initial de l'étage $t = 0$, $V_0 = \{\mathbf{0}\}$.

2. A l'étage t, $0 < t < n$, pour tout état $v_{t-1} \in V_{t-1}$ et tout élément $c_t \in \mathbb{F}_2$, il existe un état $v_t \in V_t$ tel que:

$$v_t = v_{t-1} + c_t.\boldsymbol{h}_t. \tag{1.13}$$

   La branche qui connecte l'état v_{t-1} à l'état v_t est étiquetée par le symbole c_t.

3. Au dernier étage, $t = n$, le treillis se termine sur l'état zéro final: $V_n = \{\mathbf{0}\}$.

Pour expliciter cette méthode de construction, nous allons construire un treillis pour le code binaire $(n = 4, k = 2)$ dont une matrice de contrôle $\boldsymbol{H}$ est la suivante:

$$\boldsymbol{H} = [\boldsymbol{h}_0, \boldsymbol{h}_1, \boldsymbol{h}_2, \boldsymbol{h}_3] = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \tag{1.14}$$

Le treillis associé à ce code contient $(n+1) = 5$ étages d'états V_0, V_1,...,V_4. Chaque étage contient un ensemble d'états (au maximum $2^k = 2^2 = 4$ états). Le premier étage V_0 contient un seul état, $\boldsymbol{v}_0^{(0)} = \mathbf{0}$, qui est aussi l'état de départ du treillis. De cet état émergent exactement deux branches vers deux états du deuxième étage V_1. Une branche associée à la valeur du bit 0 et l'autre associée à la valeur de bit 1. Ces deux branches arrivent respectivement sur deux états $\boldsymbol{v}_0^{(1)}$ et $\boldsymbol{v}_1^{(1)}$ du deuxième étage V_1 qui sont calculées par la formule de l'équation 1.13 tel que:

$$\boldsymbol{v}_0^{(1)} = \boldsymbol{v}_0^{(0)} + 0.\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{1.15}$$

$$\boldsymbol{v}_1^{(1)} = \boldsymbol{v}_0^{(0)} + 1.\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.16}$$

Le deuxième étage d'états contient donc les deux états $\boldsymbol{v}_0^{(1)}$ et $\boldsymbol{v}_1^{(1)}$. De chacun de ces deux états émergent aussi deux branches qui vont déterminer l'ensemble d'états du troisième étage V_2 tel que:

$$\boldsymbol{v}_0^{(2)} = \boldsymbol{v}_0^{(1)} + 0.\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{1.17}$$

$$\boldsymbol{v}_1^{(2)} = \boldsymbol{v}_1^{(1)} + 0.\boldsymbol{h}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.18}$$

$$\boldsymbol{v}_2^{(2)} = \boldsymbol{v}_0^{(1)} + 1.\boldsymbol{h}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{1.19}$$

$$\boldsymbol{v}_3^{(2)} = \boldsymbol{v}_1^{(1)} + 1.\boldsymbol{h}_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{1.20}$$

Nous répétons le même processus pour calculer les états des deux étages V_3 et V_4. La Figure 1.3 présente le treillis non élagué du code car il y a 4 états dans l'étage final.

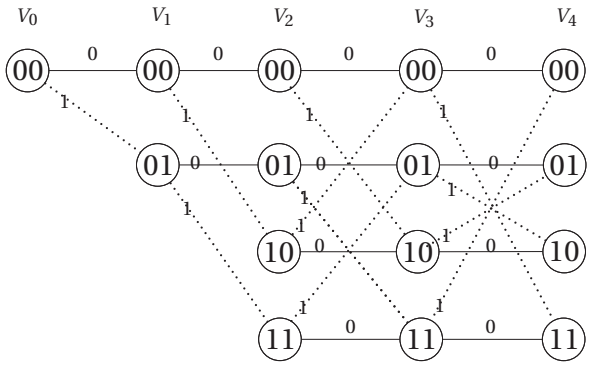

Figure 1.3. Treillis BCJR non lagu pour le code $(4,2)$.

Le treillis de la Figure 1.3 est ensuite élagué en supprimant les états non connectés à l'état final (00). La Figure 1.4 présente le treillis final élagué.

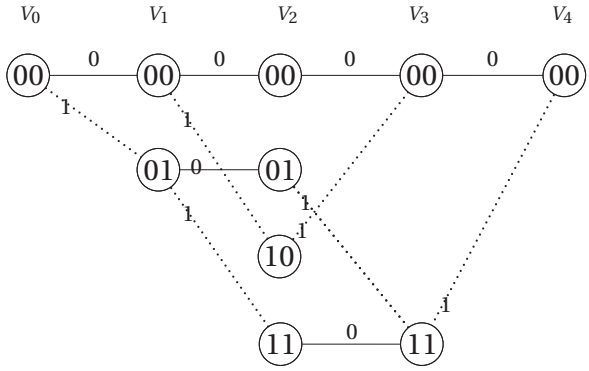

Figure 1.4. Treillis BCJR lagu pour le code $(4,2)$.

1.2.6.2 Construction par produit de treillis

Le principe de la méthode présentée par Kschischang *et al* [57] pour le calcul d'un treillis d'un code en bloc consiste à exprimer celui-ci sous forme d'une somme de sous-codes puis de calculer son treillis par le produit cartésien des treillis de ces sous-codes. Les sous-codes sont générés par les lignes de la matrice génératrice $\boldsymbol{G}$ du code et chaque sous-code est ensuite décrit par un treillis à 2 états contenant 2 chemins dont un représente le mot tout à zéro et l'autre représente la ligne elle même.

Ce paragraphe est organisé comme suit. Tout d'abord, nous allons décrire formellement le produit cartésien effectué par la méthode. Puis nous allons montrer comment les treillis élémentaires à 2 états sont calculés à partir des lignes de la matrice génératrice. La méthode est ensuite explicitée pour le calcul d'un treillis du code en bloc binaire $(4,2)$.

1.2.6.2.a Produit cartésien

Pour décrire le produit cartésien, nous considérons deux treillis $T_0 = (V^{(0)}, E^{(0)}, \mathbb{F}_2)$ et $T_1 = (V^{(1)}, E^{(1)}, \mathbb{F}_2)$, tels que: $V^{(i)} = \{V_0^{(i)}, V_1^{(i)}, \ldots, V_n^{(i)}\}$ et $E^{(i)} = \{E_0^{(i)}, E_1^{(i)}, \ldots, E_{n-1}^{(i)}\}$, pour $i \in \{0,1\}$.
Le produit cartésien entre T_0 et T_1 noté $T_0 \otimes T_1$ est défini tel que [57]:

$$T_0 \otimes T_1 = \{((v_{t-1}^0, v_{t-1}^1), (e_t^0 + e_t^1), (v_t^0, v_t^1))\}. \tag{1.21}$$

où $v_{t-1}^i \in V_{t-1}^{(i)}$, $v_t^i \in V_t^{(i)}$ et $e_t^i \in E_t^{(i)}$ pour $i \in \{0,1\}$.

1.2.6.2.b Treillis élémentaires

Dans ce paragraphe, nous allons présenter la construction des treillis élémentaires à 2 états décrivant les sous-codes générés par les lignes de la matrice génératrice du code. Pour cela, nous définissons tout d'abord la notion de "span" d'un vecteur ligne $\boldsymbol{g}_i$ par l'intervalle $[a,b]$ où a et b sont des entiers représentant respectivement la première et dernière positions non nulles du vecteur $\boldsymbol{g}_i$. Par exemple, le "span" du vecteur ligne $\boldsymbol{g}_0 = (0,0,1,0,0,0,1,0)$ est $[3,7]$. La forme d'un treillis élémentaire dépend du span du vecteur ligne qu'il représente.
Le treillis élémentaire associé à une ligne $\boldsymbol{g}_i = (g_{i0}, g_{i1}, \ldots, g_{i(n-1)})$ de "span" $[a,b]$ est calculé tel que:

- Les sections de $t = a-1$ à $t = b-1$ sont décrites par la Figure 1.5.
- Chaque section t pour $t < (a-1)$ ou $t > (b-1)$ est décrite par la Figure 1.6.

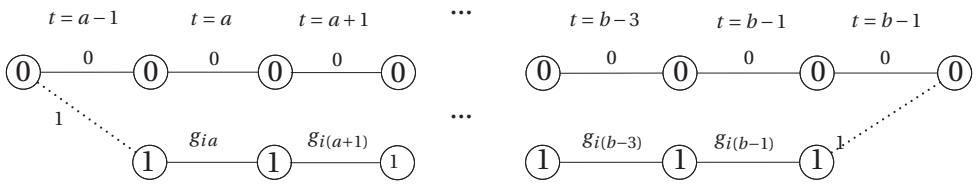

Figure 1.5. Sections du treillis lmentaires associes aux bits dans le span $[a,b]$ de la ligne $\boldsymbol{g}_i$.

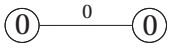

Figure 1.6. Section du treillis lmentaires associe un bit 0 en dehors du span $[a,b]$ de la ligne $\boldsymbol{g}_i$.

1.2.6.2.c Description de la méthode et exposé d'exemple

Soit $\mathscr{C}(n,k)$, un code en bloc linéaire, défini sur $\mathbb{F}_2$ et $\boldsymbol{G} = [\boldsymbol{g}_0, \boldsymbol{g}_1, \ldots, \boldsymbol{g}_{k-1}]^T$ une matrice génératrice, tel que $\boldsymbol{g}_i \in \mathbb{F}_2^n$, $0 \leqslant i \leqslant k-1$, sont les lignes de $\boldsymbol{G}$. Chaque ligne $\boldsymbol{g}_i$, $0 \leqslant i \leqslant k-1$, de $\boldsymbol{G}$ génère un sous-code de $\mathscr{C}$ noté $\langle \boldsymbol{g}_i \rangle$. Le code $\mathscr{C}$ est donc donné par:

$$\mathscr{C} = \langle \boldsymbol{g}_0 \rangle + \langle \boldsymbol{g}_1 \rangle + \ldots + \langle \boldsymbol{g}_{k-1} \rangle. \tag{1.22}$$

Soient $T_0, T_1, \ldots, T_{k-1}$ les treillis décrivant les sous-codes $\langle \boldsymbol{g}_0 \rangle, \langle \boldsymbol{g}_1 \rangle, \ldots, \langle \boldsymbol{g}_{k-1} \rangle$.
Le treillis T décrivant le code en bloc $\mathscr{C}$ est donné par le produit de ces treillis élémentaires:

$$T = T_0 \otimes T_1 \otimes \ldots \otimes T_{k-1} \tag{1.23}$$

Pour expliciter cette méthode, nous allons calculer le treillis décrivant le code en bloc binaire $(n = 4, k = 2)$ dont une matrice génératrice $\boldsymbol{G}$ est la suivante:

$$\boldsymbol{G} = \begin{bmatrix} \boldsymbol{g}_0 \\ \boldsymbol{g}_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{1.24}$$

Nous allons tout d'abord calculer les treillis associés aux sous-codes $\langle \boldsymbol{g}_0 \rangle$ et $\langle \boldsymbol{g}_1 \rangle$ générés par les 2 lignes de la matrice génératrice $\boldsymbol{G}$. Pour cela il suffit de calculer les spans des deux lignes et d'appliquer ensuite les régles décrites dans le paragraphe 1.2.6.2.b. Le span de la ligne $\boldsymbol{g}_0$ est $[1,4]$ et le span de la ligne $\boldsymbol{g}_1$ est $[2,3]$. Les treillis T_0 et T_1 décrivant les sous-codes $\langle \boldsymbol{g}_0 \rangle$ et $\langle \boldsymbol{g}_1 \rangle$ sont présentés respectivement sur les figures 1.7 et 1.8.

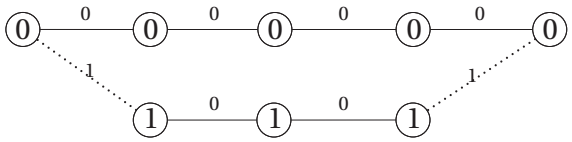

Figure 1.7. Treillis T_0 dcrivant le sous-code $\langle \boldsymbol{g}_0 \rangle$.

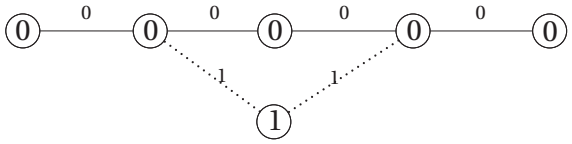

Figure 1.8. Treillis T_1 dcrivant le sous-code $\langle \boldsymbol{g}_1 \rangle$.

Le treillis T décrivant le code $(4,2)$ est donc obtenu par le produit cartésien entre les 2 treillis T_0 et T_1. Le treillis T est décrit par la Figure 1.9.

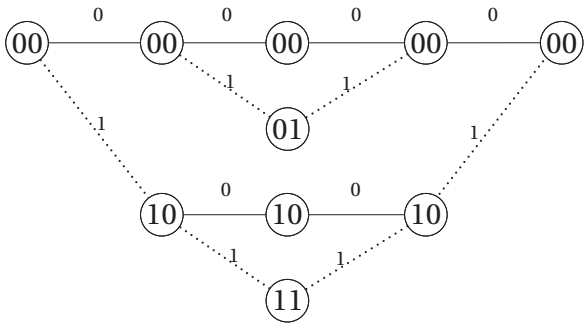

Figure 1.9. Treillis $T = T_0 \otimes T_1$ dcrivant le code $(4,2)$.

1.2.6.3 Complexité du treillis d'un code en bloc

La complexité du treillis d'un code en bloc est principalement déterminée par le nombre d'états et de branches dans le treillis [60][84]. La complexité d'état est mesurée par le nombre d'états à chaque étage du treillis et la complexité de branche par le nombre total des branches dans le treillis. Pour définir ces deux notions de complexité, nous considérons un code en bloc $\mathscr{C}(n,k)$ défini sur $\mathbb{F}_2$ et $T = (V, E, \mathbb{F}_2)$ son treillis associé tel que $V = V_0 \cup V_1 \cup \ldots \cup V_n$.

1.2.6.3.a Complexité d'état

Afin de définir formellement la complexité d'état utilisée dans la théorie des treillis, on définit tout d'abord le profil d'états $(\rho_0, \rho_1, \ldots, \rho_n)$ tel que:

$$\rho_i = log_2(|V_i|) \tag{1.25}$$

La complexité d'état $\rho_{max}(T)$ du treillis T est tout simplement donnée par:

$$\rho_{max}(T) = \max_{0 \leq i \leq n} (\rho_i) \tag{1.26}$$

Dans [35], Wolf calcule une borne supérieure de $\rho_{max}(T)$ comme suit:

$$\rho_{max}(T) \leq \max(k, n-k) \tag{1.27}$$

La complexité d'état d'un treillis permet donc de déterminer la complexité de décodage sur ce treillis. Pour l'algorithme de décodage Viterbi [33], le nombre maximum des survivants et métriques à stocker est de l'ordre de $2^{\rho_{max}(T)}$.

1.2.6.3.b Complexité de branche

La complexité de branche de T est définie par le nombre total de branches dans le treillis T. Cette complexité détermine le nombre d'opérations arithmétiques nécessaires dans un algorithme de décodage pour décoder une séquence reçue comme l'algorithme de Viterbi [33] ou BCJR [34].

1.3 Modèles des canaux de transmission

1.3.1 Canal discret sans mémoire

Un canal discret sans mémoire ou DMC (Discret Memoryless Channel) est un canal qui est caractérisé par les propriétés suivantes:

- Discret: le signal d'entée $\boldsymbol{x}$ et le signal de sortie $\boldsymbol{y}$ prennent leurs valeurs dans des alphabets finis.

- Sans mémoire: le signal de sortie $\boldsymbol{y}$ ne dépend que du signal d'entrée $\boldsymbol{x}$ et des statistiques du canal. Il ne dépend pas des signaux précédents ou suivants.

Soient $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$, une séquence émise et $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$, la séquence reçue correspondante. La probabilité conditionnelle $P(\boldsymbol{y}|\boldsymbol{x})$ est donnée par:

$$P(\boldsymbol{y}|\boldsymbol{x}) = P\{(y_0, y_1, \ldots, y_{n-1})|(x_0, x_1, \ldots, x_{n-1})\}. \tag{1.28}$$

Comme chaque échantillion y_i ne dépend que du symbole émis x_i, la probabilité $P(\boldsymbol{y}|\boldsymbol{x})$ peut donc être exprimée comme suit:

$$P(\boldsymbol{y}|\boldsymbol{x}) = \prod_{i=0}^{n-1} P(y_i|x_i). \tag{1.29}$$

1.3.2 Canal binaire à effacement

Le modèle du canal binaire à effacement ou BEC(Binary Erasure Channel) a été présenté en 1955 par Elias [10]. Les erreurs qui interviennent sur ce modèle de canal n'altèrent pas les symboles d'information mais les effacent complétement. L'entrée du canal à l'instant t, notée c_t, est un symbole binaire: $c_t \in \{0,1\}$. La sortie correspondante y_t prend ses valeurs dans l'alphabet $\{0,?,1\}$, où le symbole "?" indique une incertitude totale sur la valeur du bit. Chaque bit transmis, soit est reçu correctement ou est effacé avec une probabilité p: $y_t \in \{c_t,?\}$ et $P(y_t =?) = p$. Le graphe de transition du canal à effacement est donné par la Figure 1.10.

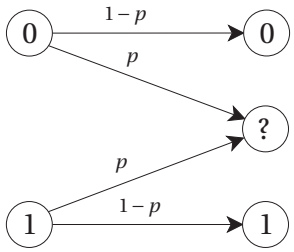

Figure 1.10. Modle du canal binaire effacement.

Le canal binaire à effacement (BEC) peut être utilisé pour modéliser la transmission sur les réseaux de données, où les paquets arrivent soient correctement, soient sont perdus à cause de la surcharge des mémoires-tampons (buffers) ou des délais importants.

1.3.3 Canal à bruit blanc additif gaussien

Le canal à bruit blanc additif gaussien ou AWGN(Additive White Gaussian Noise) est un canal à entrées discrètes et à sorties continues. Le terme "blanc" vient de l'analogie entre la lumière "blanche" qui mélange toutes les fréquences lumineuses et le bruit qui possède la même densité spectrale de puissance N_0 à toutes les fréquences. Le canal AWGN affecte le signal émis par un bruit gaussien η centré de variance σ^2 tel que : $\sigma^2 = N_0/2$. Si le signal émis $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ est décrit par la variable aléatoir X qui prend sa valeur dans un alphabet fini $\{X_0, X_1, \ldots, X_{q-1}\}$ alors le signal reçu $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$ est décrit par une variable aléatoire Y dont l'expression est donnée par : $Y = X + \eta$.

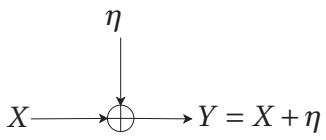

Figure 1.11. Modle du canal AWGN.

La distribution du signal reçu $\boldsymbol{y}$ est décrite par la loi de Gauss:

$$P(Y|X = X_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(\frac{Y-X_i}{\sigma})^2}. \tag{1.30}$$

Le canal à bruit blanc additif gaussien (AWGN) est le canal de référence pour étudier les performances des codes correcteurs d'erreur et leurs algorithmes de décodage. Ce modèle de canal ne tient pas compte des autres phénomènes qui peuvent affecter le signal: le fading, la sélectivité fréquentielle, interférences, etc... Le canal AWGN est utilisé aussi pour modéliser la transmission dans plusieurs systèmes de communication comme le système de communication par satellite.

1.4 Décodage des codes en bloc

Le décodage consiste à rechercher le mot de code émis $\boldsymbol{c}$ à partir du mot reçu $\boldsymbol{y}$. Il existe deux stratégies possibles pour exploiter les données reçues. La première stratégie consiste à quantifier les échantillons du signal reçu en deux niveaux seulement 0 et 1 avant de les délivrer au décodeur. Le décodage dans ce cas est dit à entrées dures. Le décodage à entrées dures qui utilise en plus la structure algébrique du code est appelé décodage algébrique. La seconde stratégie consiste à délivrer au décodeur toute l'information d'un symbole reçu quantifiée sur de multiples niveaux 4, 8, 16,... et non plus seulement avec son signe 0 ou 1. Le décodage dans ce second cas est dit à entrées souples. Le décodage à entrées souples peut apporter un gain jusqu'à 2 dB [45] par rapport au décodage à entrées dures mais il exige une complexité de calcul beaucoup plus grande. Les algorithmes qui fournissent avec la décision dure une certaine fiabilité, sont appelés algorithmes à sorties souples ou SISO (Soft-Input Soft-Output). Les algorithmes SISO ont connu, en particulier, une grande importance surtout avec l'invention des turbocodes [20] dont le décodage itératif nécéssite une information de vraisemblance sur les symboles décidés.

Dans cette section nous allons présenter quelques algorithmes de décodage à décision douce utilisés pour le décodage des codes en bloc. Cette section est organisée

comme suit. La sous-section 1.4.1 présente l'algorithme de Viterbi. La sous-section 1.4.2 présente une variante de l'algorithme de Viterbi appelée SOVA qui permet de calculer des valeurs de vraisemblances pour les symboles du mot le plus probable. La sous-section 1.4.3 présente l'algorithme BCJR qui fait un calcul exact des probabilités *a posteriori* des symboles reçus. Les sous-sections 1.4.4 et 1.4.5 présentent deux variantes de l'algorithme BCJR permenttant de réduire sa complexité. Finalement, la sous-section 1.4.6 aborde le décodage itératif. Elle présente la notion d'information extrinsèque et expose le décodage itératif à l'aide d'un exemple de turbocode.

1.4.1 Algorithme de décodage de Viterbi

Soit $\mathscr{C}(n,k)$ un code en bloc linéaire défini sur $\mathbb{F}_2$ et $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ un mot de code de $\mathscr{C}$. La modulation utilisée est la modulation à 2 phases ou BPSK qui associe le bit 0 (respectivement 1) à la valeur modulée -1(respectivement $+1$). Le mot modulé $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ associé au mot $\boldsymbol{c}$ est ensuite transmis sur un canal perturbé par un bruit blanc additif gaussien de variance $\sigma^2 = N_0/2$. Le mot reçu est noté $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$. L'objectif de l'algorithme ML (Maximum-Likelihood) est l'identification de la séquence $\hat{\boldsymbol{c}} \in \mathscr{C}$ la plus probable, sachant la séquence de symboles reçus $\boldsymbol{y}$:

$$\hat{\boldsymbol{c}} = Arg\{\max_{\boldsymbol{c}\in\mathscr{C}} P(\boldsymbol{c}/\boldsymbol{y})\} \tag{1.31}$$

d'où en utilisant la lois de Bayes:

$$\hat{\boldsymbol{c}} = Arg\{\max_{\boldsymbol{c}\in\mathscr{C}} P(\boldsymbol{y}/\boldsymbol{c})\frac{P(\boldsymbol{c})}{P(\boldsymbol{y})}\} \tag{1.32}$$

Comme tous les mots de code sont équiprobables et que la probabilité $P(\boldsymbol{y})$ est constante, on peut écrire:

$$\hat{\boldsymbol{c}} = Arg\{\max_{\boldsymbol{c}\in\mathscr{C}} P(\boldsymbol{y}/\boldsymbol{c})\} \tag{1.33}$$

Comme la probabilité $P(\boldsymbol{y}/\boldsymbol{c})$ est inférieure à 1 et que la fonction $\log(x)$ est strictement croissante alors la quantité $\log P(\boldsymbol{y}/\boldsymbol{c})$ est toujours négative. Donc maximiser la proba-

bilité $P(\boldsymbol{y}/\boldsymbol{c})$ revient à maximiser $-\log P(\boldsymbol{y}/\boldsymbol{c})$ et on peut écrire:

$$\hat{\boldsymbol{c}} = Arg\{\max_{\boldsymbol{c}\in\mathscr{C}}(-\log P(\boldsymbol{y}/\boldsymbol{c}))\} \tag{1.34}$$

Comme nous supposons que le canal est perturbé par un bruit additif de composantes indépendantes alors:

$$\begin{aligned}\hat{\boldsymbol{c}} &= Arg\{\max_{\boldsymbol{c}\in\mathscr{C}}(-\log P(\boldsymbol{y}/\boldsymbol{c}))\} \\ &= Arg\{\max_{\boldsymbol{c}\in\mathscr{C}}(-\log(\prod_{t=0}^{n-1} P(y_t|c_t)))\} \\ &= Arg\{\max_{\boldsymbol{c}\in\mathscr{C}}\sum_{t=0}^{n-1}(-\log P(y_t|c_t))\}\end{aligned} \tag{1.35}$$

Le décodage ML effectue donc une recherche exhaustive dans l'ensemble des 2^k mots de code, ce qui entraine une complexité de calcul exponentielle avec la dimension du code k.

L'algorithme de Viterbi fait une recherche du chemin le plus probable sur un treillis représentant le code [33]. Dans ce treillis, il y a une correspondance bijective entre un chemin et un mot de code. L'algorithme de Viterbi transforme donc le problème de recherche exhaustive du mot le plus probable dans l'ensemble des mots de code à une recherche du chemin le plus probable sur le treillis. Il cherche le chemin qui maximise la probabilité conjointe $P(y,S)$, où $S=(s_0,s_1,\ldots,s_n)$ désigne un chemin (séquence d'états) dans le treillis. Alors, le chemin le plus probable $\hat{S}$ est donné par:

$$\hat{S} = Arg\{\max_{S} P(y,S)\} \tag{1.36}$$

Dans la suite de ce paragraphe, nous allons tout d'abord démontrer que le chemin $\hat{S}$ représente le mot le plus probable obtenu avec le critère ML. Puis, nous allons présenter les calculs effectués par l'algorithme de Viterbi sur le treillis pour déterminer ce chemin. Comme maximiser la probabilité $P(\boldsymbol{y},S)$ revient à maximiser $-\log P(\boldsymbol{y},S)$, on peut écrire:

$$\hat{S} = Arg\{\max_{S}(-\log P(y,S))\} \tag{1.37}$$

En utilisant la lois de Bayes, l'expression du chemin le plus probable peut être donnée par:

$$\begin{aligned}\hat{S} &= Arg\{\max_{S}(-\log(P(y|S)) - \log P(S))\} \\ &= Arg\{\max_{S=(s_0,s_1,\ldots,s_n)}(-\log P((y_0,y_1,\ldots,y_{n-1}|(s_0,s_1,\ldots,s_n))) \\ &\quad -\log P(S))\}\end{aligned} \tag{1.38}$$

La quantité positive $(-\log P(S))$ est constante pour tous les chemins du treillis car ils sont équiprobables et donc elle peut être ignorée. La valeur du symbole y_t ne dépend que de la transition entre les états s_t et s_{t+1}, ce qui implique que le chemin $\hat{S}$ peut être exprimé par:

$$\begin{aligned}\hat{S} &= Arg\{\max_{S} -\log(\prod_{t=0}^{n-1} P(y_t|(s_t,s_{t+1})))\} \\ &= Arg\{\max_{S} \sum_{t=0}^{n-1}(-\log P(y_t|(s_t,s_{t+1})))\}\end{aligned} \tag{1.39}$$

Comme c_t représente l'étiquette de branche qui relie les deux états s_t et s_{t+1} de la section t du treillis, alors:

$$P(y_t|(s_t,s_{t+1})) = P(y_t|c_t) \tag{1.40}$$

Les deux équations 1.35 et 1.39 sont donc équivalentes et par conséquent le chemin le plus probable représente bien le mot le plus probable obtenu avec le critère ML. Dorénavant, nous appelons le chemin le plus probable le chemin ML.

L'algorithme de Viterbi effectue une recherche du chemin ML en parcourant le treillis de l'état initial vers l'état final. Pendant cette phase de parcours en-avant, appelée aussi forward, l'algorithme de Viterbi utilise de simples additions, comparaisons et sélections ou ACS(Add-Compare-Select) au niveau de chaque état pour écarter le chemin le moins probable appelé chemin concurrent. L'autre chemin retenu appelé aussi le chemin survivant est stocké avec l'état courant et l'état précédent pour faciliter le parcours de retour en-arrière appelé aussi backward, qui permet de retrouver le chemin ML et fournir

la séquence des symboles correspondante.

Pour décrire formellement les calculs effectués par l'algorithme de Viterbi pendant ces deux phases forward et backward, nous définissons tout d'abord quelques notations. On note par $\bar{\gamma}_t(s_t, s_{t+1})$ la métrique de la branche qui relie les deux états s_t et s_{t+1} de la section t du treillis définie par:

$$\bar{\gamma}_t(s_t, s_{t+1}) = -\log P(y_t, (s_t, s_{t+1})) \tag{1.41}$$

En utilisant la lois de Bayes,

$$\begin{aligned} \bar{\gamma}_t(s_t, s_{t+1}) &= -\log P(y_t|(s_t, s_{t+1})) - \log P((s_t, s_{t+1})) \\ &= -\log P(y_t|c_t) - \log P(c_t) \end{aligned} \tag{1.42}$$

Nous notons également par $\Gamma_t(s_t)$ la métrique du chemin survivant à l'état s_t de l'étage t du treillis, $\sigma(s_t)$ l'ensemble d'états de l'étage $t-1$ voisins de l'état s_t, et finalement par $T^{(sur)}$ une table qui permet à l'algorithme de tracer le chemin ML pendant la phase backward. La table $T^{(sur)}$ est remplie pendant la phase forward en y stockant et pour chaque état s_t de l'étage t, l'état $s_{t-1} \in \sigma(s_t)$ par lequel le chemin survivant passe à l'étage $t-1$ et l'étiquette de la branche qui relie s_{t-1} et s_t.

L'algorithme de décodage de Viterbi est résumé aux étapes suivantes:

1. **Initialisation** de la métrique du chemin Γ_t: $\Gamma_0(0) = 0$.
2. **Phase forward:**

   pour $t \leftarrow 1$ **à** n **faire**

   pour tout état s de l'étage t **faire**

   $\Gamma_t(s) \longleftarrow \max_{s' \in \sigma(s)}\{\bar{\gamma}_t(s', s) + \Gamma_{t-1}(s')\}$;

   $s^* = Arg(\max_{s' \in \sigma(s)}\{\bar{\gamma}_t(s', s) + \Gamma_{t-1}(s')\})$;

   sauvegarde de l'état s^* et l'étiquette de branche (s^*, s) dans $T^{(sur)}$;

   fin

   fin

3. **Phase backward:**

   $s = 0$; // le backward commence à l'état final 0.

   $\hat{\boldsymbol{c}} = \{\}$; // sauvegarde de la séquence ML.

   pour $t \leftarrow n$ **à** 1 **faire**

   $s^* = T^{(sur)}[t, s]$; // état du chemin ML à l'étage $t-1$

   $\hat{c}_t = T^{(sur)}[s^*, s]$; // étiquette de branche (s^*, s)

   $\hat{c} = \{\hat{c}, \hat{c}_t\}$; // on ajoute $\hat{c}_t$ à la séquence $\hat{c}$

   $s = s^*$;

   fin

Exemple: dans cet exemple, nous explicitons l'algorithme de Viterbi pour décoder le code binaire $\mathscr{C}(n = 4, k = 2)$ dont le treillis est décrit par la Figure 1.9. Nous supposons que les bits sont équiprobables, c'est-à-dire $P(c_t) = 0.5$ et les probabilités *a priori* que des 4 symboles reçus soient égaux à 0 ($P(y_t|c_t = 0)$) sont données par $(0.1, 0.2, 0.3, 0.4)$. La Figure 1.12 présente les étapes effectuées par l'algorithme de Viterbi sur le treillis décrivant le code. Pour mieux visualiser ces étapes et expliciter l'algorithme, nous avons présenté à chaque étape le treillis avec les différents calculs effectués en allant de l'étape d'initialisation ($t = 0$) à la dernière étape où l'algorithme arrive à l'état final ($t = 4$). Les valeurs sur les branches sont leurs métriques qui sont calculées à partir des probabilités *a priori* en utilisant l'équation 1.42 dans laquelle le terme $(-log(P(c_t))$ est ignoré car il est constant pour toutes les branches. Les valeurs dans les états sont les métriques des chemins survivants pour chaque état. Nous gardons les mêmes représentations des branches c'est-à-dire une branche en trait continu représente un bit 0 et une branche en trait pointillé représente un bit 1. Au niveau de chaque état le chemin en trait gras représente le chemin survivant. Le chemin ML déterminé par l'algorithme de Viterbi à l'étape finale ($t = 4$) est le chemin tout à zéro.

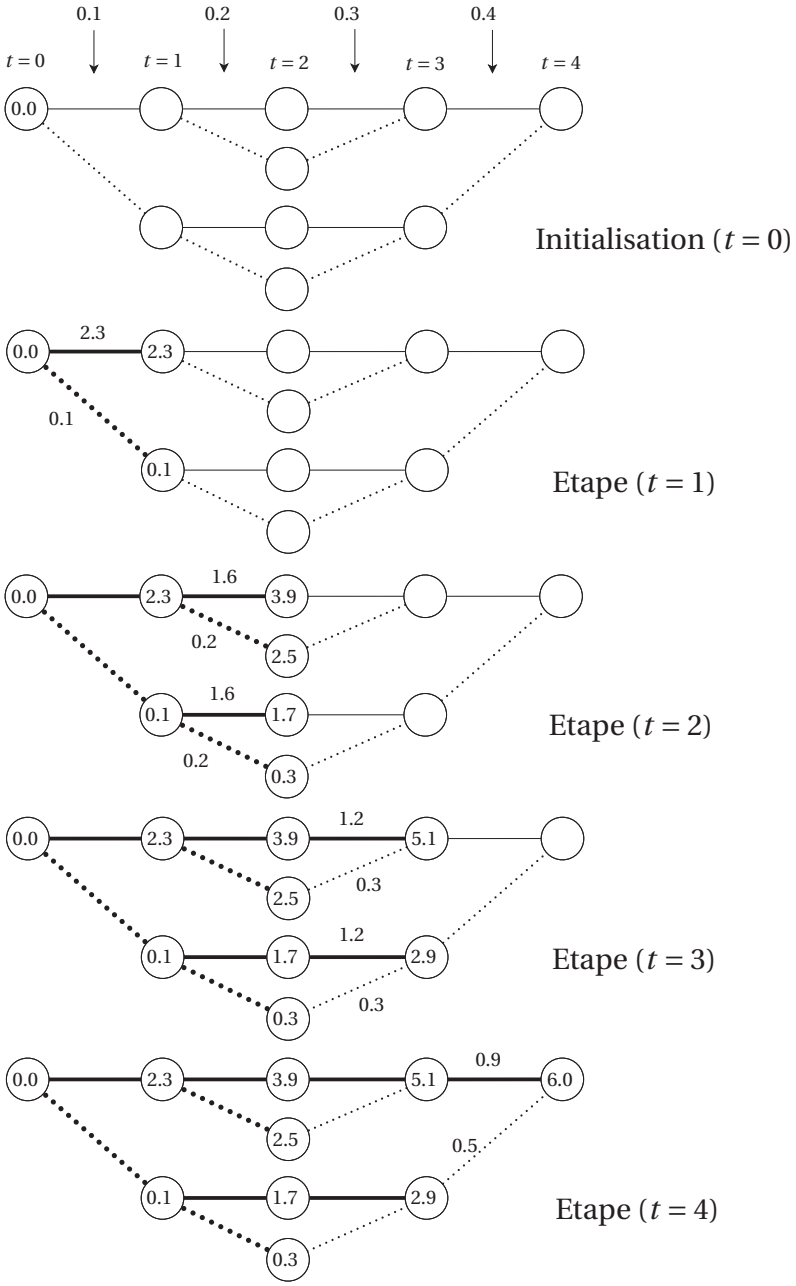

Figure 1.12. Exemple d'algorithme de Viterbi dvelopp pour le code $(4,2)$.

1.4.2 Algorithme de décodage SOVA

L'algorithme de décodage de Viterbi à sorties pondérées ou Soft-Output-Viterbi-Algorithm (SOVA) est une variante de l'algorithme de Viterbi proposée par Hagenauer en 1989 [46] qui calcule une approximation des logarithmes des rapports de vraisemblances ou LLR des bits du mot reçu.
L'algorithme SOVA apporte deux modifications à l'algorithme de Viterbi:

1. La première modification permet de fournir, en plus de la décision dure de l'algorithme de Viterbi, une fiabilité sur cette décision.
2. La deuxième modification permet à l'algorithme de Viterbi d'être utilisé dans un contexte de décodage itératif. Elle donne la possibilité d'incorporer l'information extrinsèque fournie par l'autre décodeur dans la métrique de branche donnée par l'équation 1.42. Cette information extrinsèque remplace donc la quantité $\log P(c_t)$ dans l'équation 1.42.

L'algorithme SOVA effectue une phase "forward" similaire à celle effectuée par l'algorithme de Viterbi. En plus des calculs effectués par l'algorithme de Viterbi, l'algorithme SOVA effectue les deux opérations suivantes au niveau de chaque état s_t de l'étage t du treillis:

- Sauvegarde la séquence binaire associée au chemin concurrent.
- Calcul et sauvegarde d'une fiabilité du chemin survivant notée $\Delta_t^{s_t}$ qui est la différence entre la métrique du chemin survivant et celle du chemin concurrent telle que:

$$\Delta_t^{s_t} = \max_{s_{t-1}\in\sigma(s_t)} \{\gamma_t(s_{t-1}, s_t) + \Gamma_{t-1}(s_{t-1})\} - \min_{s_{t-1}\in\sigma(s_t)} \{\gamma_t(s_{t-1}, s_t) + \Gamma_{t-1}(s_{t-1})\} \quad (1.43)$$

A la fin de la phase "forward", l'algorithme SOVA identifie le chemin ML et procéde ensuite au calcul des LLRs *a posteriori* des symboles. Pour calculer le LLR *a posteriori* du bit c_t à l'instant t du treillis, l'algorithme SOVA tient compte de tous les chemins concurrents du chemin ML après l'instant t, c'est-à-dire les chemins qui y divergent après

l'instant t du treillis, et dont la valeur de bit à l'instant t est différente de celle du bit du chemin ML. Les autres chemins qui divergent du chemin ML après l'instant t et qui donnent la même valeur de bit à l'instant t sont ignorés car ils n'ont pas d'impact sur la décision.

Nous notons par $(s_{t+1}^{ML}, s_{t+2}^{ML} \ldots, s_n^{ML})$ la séquence d'états du chemin ML se trouvant après l'étage t du treillis et par $\Delta_i^{s_i^{ML}}$, $i = t+1, \ldots, n$, la fiabilité du chemin survivant à l'état s_i^{ML} de l'étage i du treillis. On note également par c_t^i, $i = t+1, \ldots, n$, la valeur du bit, à l'instant t, du chemin concurrent du chemin ML qui y diverge à l'état s_i^{ML} de l'étage i. Ces notations sont illustrées sur le treillis décrit par la Figure 1.13. Sur ce treillis, une ligne solide représente un bit 0 et une ligne pointillée représente un bit 1. On suppose que le chemin ML est le chemin tout à zéro c'est-à-dire $s_i^{ML} = s_0$, pour $i \geq t+1$.

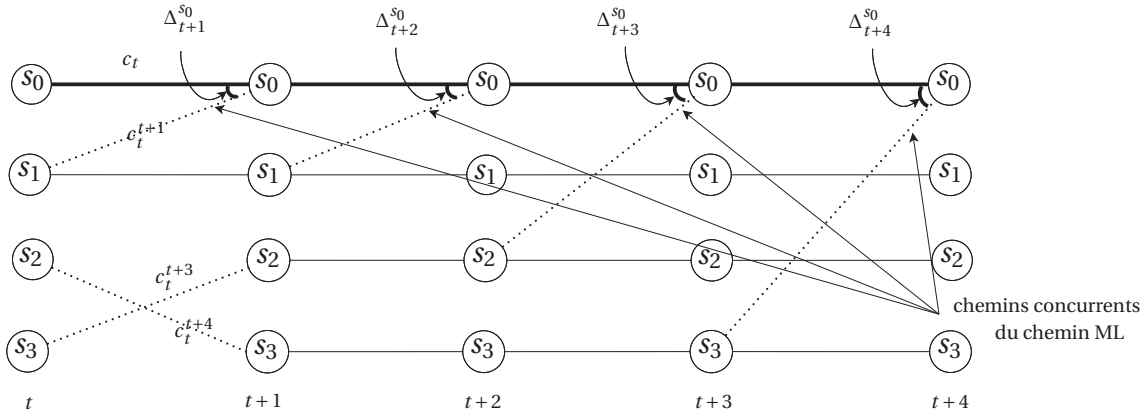

Figure 1.13. Illustration des notations.

L'algorithme SOVA estime la valeur de LLR *a posteriori* du bit c_t par:

$$LLR(c_t|\boldsymbol{y}) \approx \min_{\substack{i \geq t+1 \\ c_t \neq c_t^i}} \Delta_i^{s_i} \tag{1.44}$$

Exemple: pour mieux expliquer le calcul du $LLR(c_t|\boldsymbol{y})$ effectué par l'algorithme SOVA, nous prenons l'exemple du treillis présenté sur la Figure 1.14. Nous supposons que le chemin tout à zéro est le chemin ML et il est représenté en gras noir. Nous allons mon-

trer comment l'algorithme SOVA calcule le LLR du bit c_{n-4}. Tout d'abord, nous allons identifier les chemins qui divergent du chemin ML et dont les valeurs des bits c^i_{n-4}, $i = n-3,\ldots,n$, sont différentes de la valeur du bit c_{n-4} du chemin ML ($c_{n-4} = 0$). Les chemins identifiés sont les trois chemins sur le treillis qui sont doublés et surlignés en vert, rouge et bleu.

Donc, le LLR du bit c_{n-4} est donné par:

$$LLR(c_{n-4}|\boldsymbol{y}) \approx \min(\Delta^{s_0}_{n-3}, \Delta^{s_0}_{n-1}, \Delta^{s_0}_{n}) \tag{1.45}$$

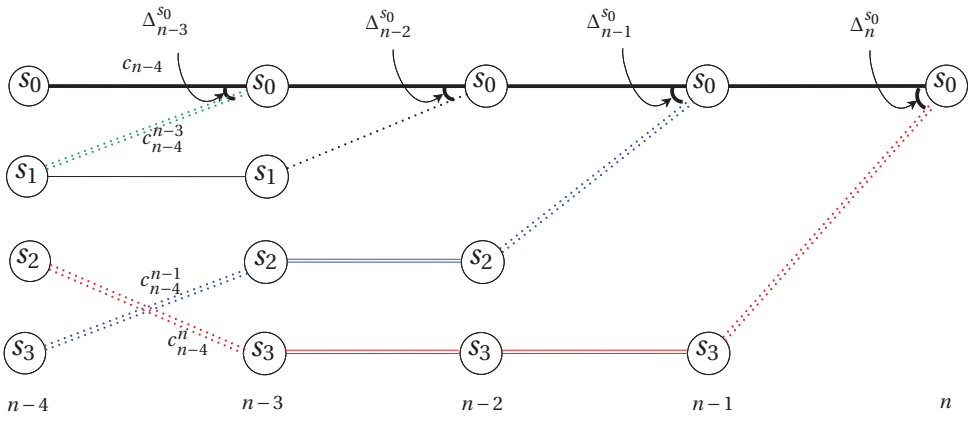

Figure 1.14. Exemple de calcul de LLR *a posteriori* du bit c_{n-4}.

1.4.3 Algorithme de décodage BCJR

L'algorithme BCJR [34] effectue le calcul des probabilités *a posteriori* des symboles sachant la séquence reçue.

Nous notons par $\mathscr{C}(n,k)$ un code en bloc linéaire défini sur $\mathbb{F}_2$ et $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ un mot de code de $\mathscr{C}$. Le mot de code $\boldsymbol{c}$ est modulé par une modulation à deux phases ou BPSK. Le mot modulé $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ associé au mot $\boldsymbol{c}$ est ensuite transmis sur un canal que nous supposons perturbé par un bruit AWGN de variance $\sigma^2 = N_0/2$. Le mot reçu est noté $y = (y_0, y_1, \ldots, y_{n-1})$.

L'algorithme BCJR effectue le calcul de la probabilité *a posteriori* $P(c_t|y)$ du bit c_t, à

l'aide du treillis décrivant le code $\mathscr{C}$. Dans ce treillis, tout chemin représente un mot de code et tout mot de code est représenté par un chemin. Ceci implique aussi que, tout bit c_t du mot de code émis $\boldsymbol{c}$ est représenté par une transision (branche) d'un état s_t de l'étage t à un état s_{t+1} de l'étage $(t+1)$. Donc, calculer la probabilité que le bit émis c_t soit égal à u, $u = 0,1$, sachant la séquence reçue $\boldsymbol{y}$ revient exactement à calculer la probabilité que la transision entre les deux étages du treillis t et $(t+1)$ soit assurée par une branche d'étiquette u. Comme à l'instant t du treillis, une seule transition peut se produire donc la probabilité d'une transition est la somme des probabilités individuelles. Ceci peut être traduit formellement par:

$$P(c_t = u|\boldsymbol{y}) = \sum_{(s_t,s_{t+1})=u} P(s_t, s_{t+1}|\boldsymbol{y}). \tag{1.46}$$

En utilisant la loi de Bayes, la probabilité $P(c_t = u|\boldsymbol{y})$ peut être exprimée telle que:

$$P(c_t = u|\boldsymbol{y}) = \sum_{(s_t,s_{t+1})=u} P(s_t, s_{t+1}, \boldsymbol{y})/P(\boldsymbol{y}) \tag{1.47}$$

Comme la probabilité $P(\boldsymbol{y})$ est constante pour tous les mots de code, l'expression précédente de la probabilité *a posteriori* peut donc être simplifiée en ignorant $P(\boldsymbol{y})$ telle que:

$$P(c_t = u|\boldsymbol{y}) = \sum_{(s_t,s_{t+1})=u} P(s_t, s_{t+1}, \boldsymbol{y}) \tag{1.48}$$

Pour calculer la probabilité $P(c_t = u|\boldsymbol{y})$, il suffit de calculer les probabilités conjointes $P(s_t, s_{t+1}, \boldsymbol{y})$. Si on note par $\boldsymbol{y}_{t_0}^{t_1}$ la partie de $\boldsymbol{y}$ reçue entre les instants t_0 et t_1 alors le mot $\boldsymbol{y}$ peut être décomposé en 3 parties à l'instant t: la séquence reçue avant l'instant t, $\boldsymbol{y}_0^{t-1}$, le symbol reçu à l'instant t, y_t, et la séquence reçue après l'instant t, $\boldsymbol{y}_{t+1}^{n-1}$. En utilisant la loi de Bayes, la probabilité conjointe $P(s_t, s_{t+1}, \boldsymbol{y})$ peut être exprimée par:

$$\begin{aligned} P(s_t, s_{t+1}, \boldsymbol{y}) &= P(s_t, s_{t+1}, \boldsymbol{y}_0^{t-1}, y_t, \boldsymbol{y}_{t+1}^{n-1}) \\ &= P(\boldsymbol{y}_{t+1}^{n-1}|s_t, s_{t+1}, \boldsymbol{y}_0^{t-1}, y_t) P(s_t, s_{t+1}, \boldsymbol{y}_0^{t-1}, y_t) \\ &= P(\boldsymbol{y}_{t+1}^{n-1}|s_t, s_{t+1}, \boldsymbol{y}_0^{t-1}, y_t) P(y_t, s_{t+1}|s_t, \boldsymbol{y}_0^{t-1}) P(s_t, \boldsymbol{y}_0^{t-1}) \end{aligned} \tag{1.49}$$

Sachant que la séquence reçue $\boldsymbol{y}_{t+1}^{n-1}$ après l'instant t ne dépend pas ni de l'état précédent s_t, ni de la séquence précédente $\boldsymbol{y}_0^{t-1}$ et ni du symbole y_t et que la séquence $\boldsymbol{y}_0^{t-1}$ ne dépend pas du symbole y_t ni de l'état s_{t+1}, alors la probabilité $P(s_t, s_{t+1}, \boldsymbol{y})$ peut donc être simplifiée telle que:

$$\begin{aligned} P(s_t, s_{t+1}, \boldsymbol{y}) &= \underbrace{P(\boldsymbol{y}_{t+1}^{n-1}|s_{t+1})}_{\beta_{t+1}(s_{t+1})} \underbrace{P(y_t, s_{t+1}|s_t)}_{\gamma_t(s_t,s_{t+1})} \underbrace{P(s_t, \boldsymbol{y}_0^{t-1})}_{\alpha_t(s_t)} \\ &= \alpha_t(s_t)\gamma_t(s_t, s_{t+1})\beta_{t+1}(s_{t+1}). \end{aligned} \tag{1.50}$$

Les quantités α_t, β_t, et γ_t sont appelées respectivement probabilité d'état "forward", probabilité d'état "backward" et la probabilité de branche. Les probabilités "forward" et "backward" peuvent être calculées récursivement telles que:

$$\alpha_t(s_t) = \sum_{s_{t-1} \in \sigma_{t-1}(s_t)} \gamma_t(s_{t-1}, s_t).\alpha_{t-1}(s_{t-1}), \text{ avec } \alpha_0(s_0) = 1. \tag{1.51}$$

$$\beta_{t+1}(s_{t+1}) = \sum_{s_{t+2} \in \sigma_{t+2}(s_{t+1})} \gamma_{t+1}(s_{t+1}, s_{t+2}).\beta_{t+2}(s_{t+2}), \text{ avec } \beta_n(s_n) = 1. \tag{1.52}$$

où $\sigma_{t-1}(s_t)$ (resp. $\sigma_{t+2}(s_{t+1})$) est l'ensemble d'états, de l'étage $t-1$ (resp. $t+2$), voisins de l'état s_t (resp. s_{t+1}) de l'étage t (resp. $t+1$) du treillis.

La probabilité de branche $\gamma_t(s_{t-1}, s_t)$ peut être exprimée en fonction des probabilités *a priori* des bits émis et reçu comme suit:

$$\gamma_t(s_{t-1}, s_t) = P(y_t|x_t)P(x_t) \tag{1.53}$$

Les détails des calculs des probabilités α_t, β_t et γ_t sont dans l'annexe *A*.

Finalement la probabilité *a posteriori* du bit c_t est donnée par:

$$P(c_t = u|\boldsymbol{y}) = \sum_{(s_t, s_{t+1})=u} \alpha_t(s_t)\gamma_t(s_t, s_{t+1})\beta_{t+1}(s_{t+1}). \tag{1.54}$$

Les étapes effectuées par l'algorithme BCJR pour calculer la probabilité *a posteriori* $P(c_t = u|\boldsymbol{y})$ sur le treillis décrivant le code en bloc $\mathscr{C}$ sont:

1. Initialisation des probabilités forward $\alpha_0(s_0) = 1$ et backward $\beta_n(s_n) = 1$.

2. Calcul des probabilités des branches $\gamma_t(s_t, s_{t+1})$, $t = 0, 1, \ldots, n-1$, en utilisant l'équation 1.53.
3. Calcul des probabilités forward α_t, $t = 1, 2, \ldots, n-1$, en parcourant le treillis de l'état initial vers l'état final et en utilisant l'équation 4.15.
4. Calcul des probabilités backward β_t, $t = n-1, n-2, \ldots, 1$, en parcourant le treillis de l'état final vers l'état initial et en utilisant l'équation 4.16.
5. Calcul de la probabilité *a posteriori* $P(c_t = u|\boldsymbol{y})$ du bit c_t en utilisant l'équation 1.54 et les valeurs des probabilités α_t et β_t calculées dans les étapes 3 et 4.

Pour la simplicité des calculs et la clarté de l'exposé des autres variantes de BCJR présentées dans les deux paragraphes qui suivent, nous définissons le log de rapport de vraisemblance *a posteriori* du bit c_t par:

$$LLR(c_t|\boldsymbol{y}) = \log\left[\frac{P(c_t = 1|\boldsymbol{y})}{P(c_t = 0|\boldsymbol{y})}\right] \tag{1.55}$$

Ce rapport peut donc être exprimé en fonction des probabilités α_t, β_t et γ_t précédentes tel que:

$$LLR(c_t|y) = \log\left[\frac{\sum_{(s_t,s_{t+1})=1} \alpha_t(s_t).\gamma_t(s_t, s_{t+1}).\beta_{t+1}(s_{t+1})}{\sum_{(s_t,s_{t+1})=0} \alpha_t(s_t).\gamma_t(s_t, s_{t+1}).\beta_{t+1}(s_{t+1})})\right] \tag{1.56}$$

L'algorithme BCJR dans sa façon présentée précédemment est énormément couteux en termes de calcul à faire. Dans les deux paragraphes qui suivent, nous allons présenter deux variantes simplifiées de l'algorithme BCJR: Max-Log-MAP et Log-MAP.

1.4.4 Algorithme de décodage Max-Log-MAP

L'algorithme Max-Log-MAP [47] est une variante simplifiée de l'algorithme BCJR. Il réduit considérablement la charge combinatoire engendrée par l'algorithme BCJR en calculant des approximations des probabilités forward (α_t) et backward (β_t) à l'aide d'une approximation de la fonction max* définie dans $\mathbb{R}$ par:

$$\max^*(z_0, z_1, \ldots, z_{q-1}) \triangleq \log\left(\sum_{i=0}^{q-1} e^{z_i}\right) \tag{1.57}$$

où $z_0, z_1, \ldots, z_{q-1}$ sont des réels.

La fonction max* peut être calculée récursivement comme suit:

$$1)\ z = \log\left(\underbrace{e^{z_0} + e^{z_1}}_{e^{v_0}} + \ldots + e^{z_{q-1}}\right) \Rightarrow v_0 = \log(e^{z_0} + e^{z_1}) = \max(z_0, z_1) + f_c(|z_0 - z_1|)$$

$$2)\ z = \log\left(\underbrace{e^{v_0} + e^{z_2}}_{e^{v_1}} + \ldots + e^{z_{q-1}}\right) \Rightarrow v_1 = \log(e^{v_0} + e^{z_2}) = \max(v_0, z_2) + f_c(|v_0 - z_2|)$$

$$\vdots$$

$$q-2)\ z = \log\left(e^{v_{q-2}} + e^{z_{q-1}}\right) = \log(e^{v_{q-2}} + e^{z_{q-1}}) = \max(e^{v_{q-2}}, e^{z_{q-1}}) + f_c(|e^{v_{q-2}} - e^{z_{q-1}}|)$$

où f_c est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ définie par:

$$f_c \left| \begin{array}{ccc} \mathbb{R} & \rightsquigarrow & \mathbb{R} \\ x & \longmapsto & f_c(x) = \log(1 + e^{-x}) \end{array} \right.$$

Nous appelons le terme calculé par la fonction f_c le terme correctif.

L'algorithme Max-Log-MAP fait une approximation de la fonction max* en négligeant le terme correctif f_c:

$$\max^*(z_0, z_1, \ldots, z_{q-1}) \approx \max_{0 \leq i \leq q-1}(z_i) \tag{1.58}$$

Pour pouvoir exploiter cette approximation, nous définissons les variables $A_t(s_t)$, $B_t(s_t)$ et $\chi_t(s_t, s_{t+1})$ telles que:

$$A_t(s_t) \triangleq \log(\alpha_t(s_t)), \text{ avec } A_0(s_0) = 0 \tag{1.59}$$

$$B_{t+1}(s_{t+1}) \triangleq \log(\beta_{t+1}(s_{t+1})), \text{ avec } B_n(s_n) = 0 \tag{1.60}$$

$$\chi_t(s_t, s_{t+1}) \triangleq \log(\gamma_t(s_t, s_{t+1})) \tag{1.61}$$

En utilisant l'expression récursive de $\alpha_t(s_t)$ dans l'équation 4.15 et l'approximation dans l'équation 1.58, $A_t(s_t)$ est donné par:

$$\begin{aligned} A_t(s_t) &\triangleq \log(\alpha_t(s_t)) \\ &= \log\left(\sum_{s_{t-1}\in\sigma_{t-1}(s_t)} \gamma_t(s_{t-1},s_t).\alpha_{t-1}(s_{t-1})\right) \\ &= \log\left(\sum_{s_{t-1}\in\sigma_{t-1}(s_t)} e^{(\chi_t(s_{t-1},s_t)+A_{t-1}(s_{t-1}))}\right) \\ &= \max^*_{s_{t-1}\in\sigma_{t-1}(s_t)}(\chi_t(s_{t-1},s_t)+A_{t-1}(s_{t-1})) \\ &\approx \max_{s_{t-1}\in\sigma_{t-1}(s_t)}(\chi_t(s_{t-1},s_t)+A_{t-1}(s_{t-1})) \end{aligned} \tag{1.62}$$

L'algorithme Max-Log-MAP effectue exactement les même calculs effectués par l'algorithme de Viterbi dans sa phase forward.

La valeur de la métrique $B_{t+1}(s_{t+1})$ est calculée de la même façon que dans l'équation 1.62 en parcourant le treillis de l'état final vers l'état initial:

$$\begin{aligned} B_{t+1}(s_{t+1}) &\triangleq \log(\beta_{t+1}(s_{t+1})) \\ &= \max^*_{s_{t+2}\in\sigma_{t+2}(s_{t+1})}(\chi_{t+1}(s_{t+1},s_{t+2})+B_{t+2}(s_{t+2})) \\ &\approx \max_{s_{t+2}\in\sigma_{t+2}(s_{t+1})}(\chi_{t+1}(s_{t+1},s_{t+2})+B_{t+2}(s_{t+2})) \end{aligned} \tag{1.63}$$

Le calcul de la métrique $B_{t+1}(s_{t+1})$ dans l'équation 1.63 est aussi similaire au calcul effectué par l'algorithme de Viterbi mais dans le sens inverse ou backward. C'est pourquoi l'algorithme Max-Log-MAP est vu comme un algorithme de Viterbi appliqué dans les deux sens du treillis (forward et backward).

La valeur de LLR du bit $LLR(c_t|\boldsymbol{y})$ calculée par l'algorithme Max-Log-MAP est une approximation de l'équation 1.56 telle que:

$$
\begin{aligned}
LLR(c_t|\boldsymbol{y}) &= \log\left[\frac{\sum_{(s_t,s_{t+1})=1}\alpha_t(s_t).\gamma_t(s_t,s_{t+1}).\beta_{t+1}(s_{t+1})}{\sum_{(s_t,s_{t+1})=0}\alpha_t(s_t).\gamma_t(s_t,s_{t+1}).\beta_{t+1}(s_{t+1})})\right] \\
&= \max^*_{(s_t,s_{t+1})=1}(A_t(s_t)+\chi_t(s_t,s_{t+1})+B_{t+1}(s_{t+1}))- \\
&\quad \max^*_{(s_t,s_{t+1})=0}(A_t(s_t)+\chi_t(s_t,s_{t+1})+B_{t+1}(s_{t+1})) \\
&\approx \max_{(s_t,s_{t+1})=1}(A_t(s_t)+\chi_t(s_t,s_{t+1})+B_{t+1}(s_{t+1}))- \\
&\quad \max_{(s_t,s_{t+1})=0}(A_t(s_t)+\chi_t(s_t,s_{t+1})+B_{t+1}(s_{t+1})) \qquad (1.64)
\end{aligned}
$$

Pour calculer la quantité $A_t(s_t)+\chi_t(s_t,s_{t+1})+B_{t+1}(s_{t+1})$, l'algorithme Max-Log-MAP considère seulement le meilleur chemin arrivant à l'étage t du treillis au lieu de considérer tous les chemins comme le fait l'algorithme BCJR. Cette approximation est la raison pour la sous-optimalité de l'algorithme Max-Log-MAP comparé à l'algorithme BCJR.

Les étapes effectuées par l'algorithme Max-Log-MAP pour calculer LLR *a posteriori* $LLR(c_t|\boldsymbol{y})$ du bit c_t sur le treillis décrivant le code en bloc $\mathscr{C}$ sont données par:

1. Initialisation des métriques forward $A_0(s_0)=0$ et backward $B_n(s_n)=0$.
2. Calcul des métriques des branches $\chi_t(s_t,s_{t+1})$, $t=0,1,\ldots,n-1$, en utilisant l'équation 1.61.
3. Calcul des métriques forward A_t, $t=1,2,\ldots,n-1$, en parcourant le treillis de l'état initial vers l'état final et en utilisant l'équation 1.62.
4. Calcul des métriques backward B_t, $t=n-1,n-2,\ldots,1$, en parcourant le treillis de l'état final vers l'état initial et en utilisant l'équation 1.63.
5. Calcul du LLR *a posteriori* $LLR(c_t|\boldsymbol{y})$ du bit c_t en utilisant l'équation 1.64 et les valeurs des probabilités A_t et B_t calculées dans les étapes 3 et 4.

1.4.5 Algorithme de décodage Log-MAP

Au contraire de l'algorithme Max-Log-MAP qui fait une approximation des probabilités α_t et β_t, l'algorithme Log-MAP [48] effectue leur calcul exact en gardant le terme correctif calculé par f_c qui a été négligé par l'algorithme Max-Log-MAP. L'algorithme Log-MAP conserve le terme correctif f_c mais d'une manière économe en transformant son calcul à une simple lecture de mémoire ou ROM (Read-Only Memory) dans une table de correspondance (en anglais look-up table) de 8 valeurs. L'algorithme Log-MAP réalise les mêmes performances que le BCJR.

Nous allons tout d'abord montrer pourquoi les valeurs du terme correctif f_c sont limitées à seulement 8 valeurs. Nous pouvons répondre à cette question en observant la courbe des valeurs de f_c en fonction de la variable x présenée sur Figure 1.15. Nous constatons que la valeur maximale de f_c est petite (égale log(2) = 0.693) et que si x augmente la fonction f_c décroit rapidement vers zéro. Pour $x \geq 5$, la valeur de $f_c(x)$ est quasiment nulle. Cette réalité est exploitée par l'algorithme Log-MAP pour quantifier les valeurs de f_c sur 8 valeurs calculées pour x entre 0 et 5 et stockées dans une table de correspondance (look-up table) [48]. Une représentation plus fine par l'augmentation du nombre de valeurs dans la table n'améliore pas les performances [48].

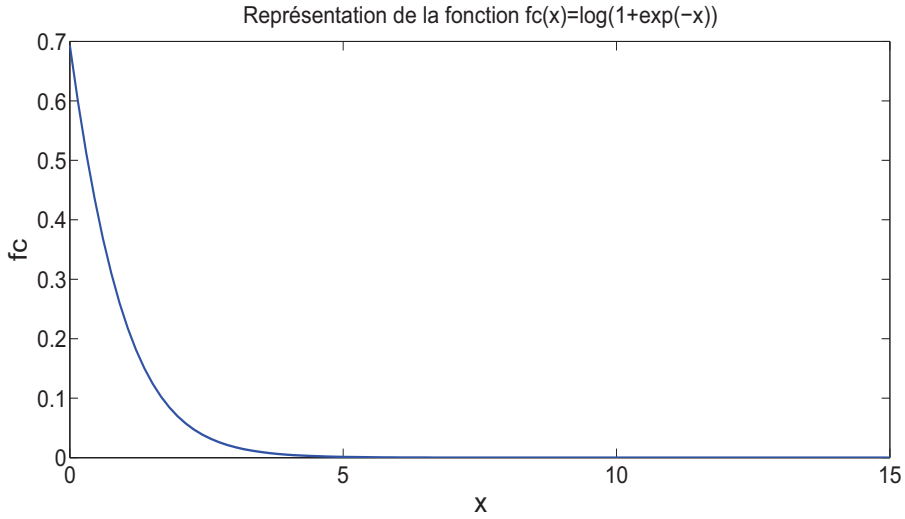

Figure 1.15. Reprsentation de la fonction $f_c = \log(1 + e^{-x})$ sur $\mathbb{R}^+$.

Donc pour résumer, l'algorithme Log-MAP effectue les mêmes calculs que l'algorithme Max-Log-MAP sauf qu'il calcule les métriques $A_t(s_t)$ et $B_t(s_t)$ en gardant le terme correctif f_c et en utilisant la table de correspondance pour son calcul.
L'algorithme Log-MAP effectue le même nombre d'opérations que l'algorithme Max-Log-MAP sauf qu'il effectue en plus une lecture de mémoire ou ROM (Read-Only Memory) pour y extraire la valeur du terme correctif.
Nous notons finalement qu'il existe des techniques permettant de calculer une approximation du terme correctif f_c sans avoir besoin de la table de correspondance mais cette approximation engendre une dégradation légère des performances [65][66].

1.4.6 Décodage itératif

Le décodage itératif consiste à exécuter le processus de décodage en plusieurs étapes ou itérations simples. Pendant chaque étape, deux ou plusieurs décodeurs élémentaires échangent des informations sur les bits du mot reçu afin de pouvoir fournir une décision plus fiable sur ces bits. L'information échangée est appelée information extrinsèque.

1.4.6.1 Information extrinsèque

Comme le LLR, l'information extrinsèque mesure aussi une fiabilité sur le bit décidé. Dans un processus de décodage itératif, chaque décodeur calcule la valeur de l'information extrinsèque d'un bit sans l'influence de l'observation du canal ou les informations fournies par les autres décodeurs sur ce même bit. Cela évite la corrélation qui peut être causée par le retour, à un décodeur, d'une information qu'il a déjà générée dans une étape précédente du processus.
Le décodeur traite l'information extrinsèque selon l'une des deux approches suivantes [67]:

1. Le décodeur traite l'information extrinsèque comme étant la sortie d'un canal à bruit blanc gaussien. Dans ce cas, l'information extrinsèque z_t du bit c_t est sup-

posée telle que:

$$z_t = \eta_t x_t + n'_t \tag{1.65}$$

où n'_t est une variable aléatoire réelle gaussienne de moyenne nulle et de variance σ_z, $\eta_t \triangleq |E(z_t|x_t)|$ et x_t est la valeur modulée de c_t avec la modulation BPSK. Les valeurs de η_t et σ_z sont estimées pour chaque mot et chaque itération.

2. Le décodeur traite l'information extrinsèque comme une information *a priori* qui va être utilisée dans l'étape suivante de l'algorithme. Le décodeur estime z_t comme étant le LLR des probabilités a priori du bit c_t:

$$z_t \approx \log \frac{P(c_t = 1)}{P(c_t = 0)}. \tag{1.66}$$

Les probabilités a priori peuvent donc exprimées comme suit:

$$P(c_t = \pm 1) \approx \frac{e^{c_t z_t}}{1 + e^{c_t z_t}} \tag{1.67}$$

L'avantage de cette approche est qu'elle ne demande pas une estimation des paramètres η_t et σ_z.

Il a été démontré dans [67] que l'utilisation de la deuxième approche dans le décodage des turbocodes donne de meilleures performances que la première approche. Dans le cadre de nos travaux, nous avons utilisé donc cette deuxième approche.
Dans un schéma de décodage itératif, les décodeurs élémentaires utilisent des algorithmes de décodage à décision douce SISO (Soft-Input-Soft-Output). Par exemple, un décodeur qui utilise l'algorithme BCJR incorpore l'information extrinsèque fournie par le(s) autre(s) décodeur(s) dans le calcul de la métrique de branche dans l'équation 1.53. L'information extrinsèque $L_e(c_t)$ peut être déduite de la valeur du LLR a posteriori, $LLR(c_t|y)$ par:

$$L_e(c_t) = LLR(c_t|y) - L(c_t) - L_c y_t. \tag{1.68}$$

A la première itération $L(c_t) = 0$ avec l'hypothèse que les bits sont équiprobables. A l'itération it, la valeur de $L(c_t)$ est l'information extrinsèque fournie par le(s) autre(s)

décodeurs à l'itération précédente $it-1$.

Dans le cas où l'algorithme utilisé par le décodeur est un algorithme BCJR, la valeur de $L_e(c_t)$ peut être calculée directement comme suit:

$$L_e(c_t) = \log\left[\frac{\sum_{(s_t,s_{t+1})=1} \alpha_t(s_t).\beta_{t+1}(s_{t+1})}{\sum_{(s_t,s_{t+1})=0} \alpha_t(s_t).\beta_{t+1}(s_{t+1})})\right] \tag{1.69}$$

Si le décodeur utilise l'algorithme Max-Log-MAP, l'information extrinsèque L_e peut être aussi calculée directement par:

$$L_e(c_t) \approx \max_{(s_t,s_{t+1})=1}(A_t(s_t)+B_{t+1}(s_{t+1})) - \max_{(s_t,s_{t+1})=0}(A_t(s_t)+B_{t+1}(s_{t+1})) \tag{1.70}$$

Dans un décodage itératif, les performances de l'algorithme Max-Log-MAP peuvent être améliorées en utilisant un coefficient qui pondère les informations extrinsèques echangées entre les codes de base. Le choix convenable de ce coefficient peut améliorer les performances de l'algorithme Max-Log-MAP de 0.4 dB [62–64].

1.4.6.2 Exemple de décodage itératif

Dans cette partie, nous présentons un exemple de décodage itératif à l'aide des turbocodes [20]. La structure originele de codage des turbocodes est composée de deux codes convolutifs en parallèle séparés par une permutation π comme la montre la Figure 1.16.

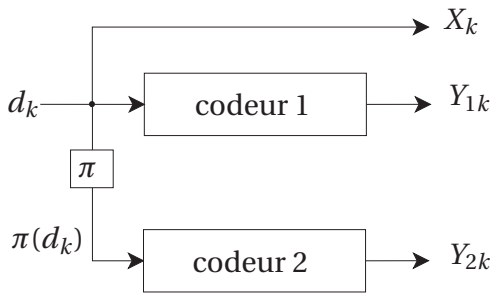

Figure 1.16. Structure de codage des turbocodes.

où $\{d_k\}$ est la séquence de bits d'informations, $\{\pi(d_k)\}$ est la séquence $\{d_k\}$ permutée par la permutation π, Y_{1k} et Y_{2k} sont les parties de redondances délivrés respectivement par les codeurs 1 et 2.
La structure de décodage associée au codeur précédent est donnée par la Figure 1.17.

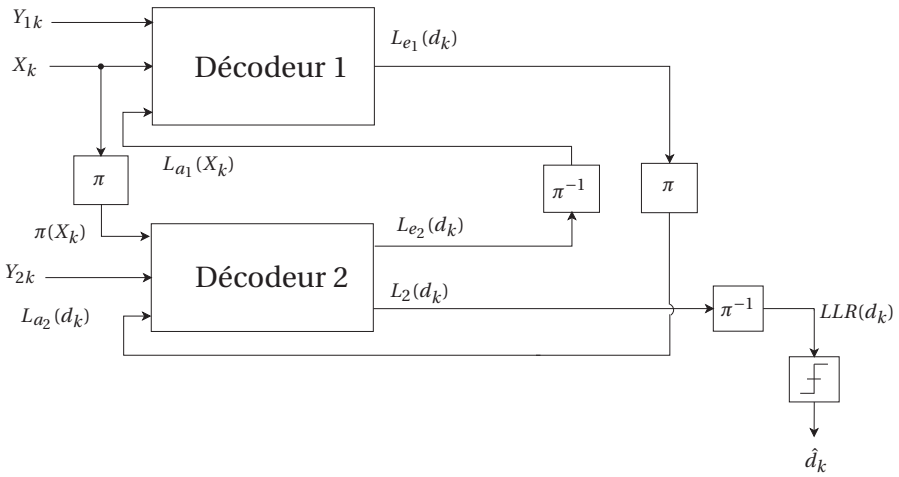

Figure 1.17. Dcodeur des turbocodes.

Le décodage itératif dont la structure est décrite sur la Figure 1.17 suit les étapes suivantes:

1. Initialisation des *a priori* pour le décodeur $L_{a_1}(X_k) = 0$ pour $k = 0,1,2,\ldots$. Aller à l'étape suivante.

2. Le décodeur 1, disposant le vecteur des *a priori* $L_{a_1}(X_k)$ et du mot reçu du canal Y_{1k}, applique un algorithme de décodage à décision douce (BCJR ou autre) et fournit ensuite une information extrinsèque $L_{e_1}(d_k)$ sur chaque bit d_k calculée par l'équation 1.68;

3. L'information extrinsèque $L_{e_1}(d_k)$ est permutée par la permutation π et ensuite délivrée au décodeur 2 comme *a priori* $L_{a_2}(X_k)$. Le décodeur 2, disposant de l'information *a priori* et du mot reçu du canal, applique un algorithme de décodage à décision douce (BCJR ou autre) et fournit ensuite une information ex-

trinsèque $L_{e_2}(d_k)$ sur chaque bit X_k. Cette information, après son permutation par π^{-1}, sera délivrée au décodeur 1 comme a priori $L_{a_1}(X_k)$. Si le nombre maximal d'itérations fixé est atteint, aller à l'étape suivante. Sinon, aller à l'étape 2.

4. Après un certain nombre d'itérations, le LLR *a posteriori* $L_2(d_k)$ à la sortie du décodeur 2 est permuté par π^{-1} et délivré comme le LLR du bit $LLR(d_k)$. Une décision dure est prise sur la valeur de bit selon le signe de son LLR.

1.5 Conclusion du chapitre 1

Dans ce chapitre, nous avons tout d'abord présenté un modèle d'une chaine de transmission numérique dans laquelle nous avons montré la position du codage et décodage. Nous avons donné ensuite des notions de base sur les codes en bloc. Les codes en bloc peuvent être représentés de plusieurs manière. La représentation en treillis d'un code en bloc conduit souvent à la conception des algorithmes de décodage de bonnes performances. Deux méthodes de construction d'un treillis d'un code en bloc ont été présentées dans ce chapitre: la méthode Wolf-BCJR qui utilise la matrice de contrôle du code et la méthode de produit des treillis qui utilise la matrice génératrice. Les treillis des codes représentent des supports pour plusieurs algorithmes de décodage. Certains algorithmes liés à la structure en treillis du code ont été ensuite présentés. Nous avons présenté l'algorithme de Viterbi qui cherche le mot le plus probable sur le treillis décrivant le code. Puis l'algorithme SOVA qui est une variante de l'algorithme de Viterbi qui calcule des vraisemblances pour les symboles du mot le plus probable sur le treillis. Nous avons ensuite présenté l'algorithme BCJR qui effectue le calcul des probabilités *a posteriori* des bits reçus. Deux variantes moins complexes de l'algorithme BCJR ont été aussi présentées: Max-Log-MAP et Log-MAP. Le chapitre a été clôturé par la présentation du décodage itératif. Nous avons présenté la notion d'information extrinsèque et donné deux approches permettant de l'exploiter dans un contexte itératif. Puis nous avons expliciter le décodage itératif à l'aide d'un exemple des turbocodes.

Chapitre 2

LES CODES CORTEX

2.1 Introduction

Les codes Cortex ont été inventés en 1998 par Carlach et Vervoux [51] et constituent une famille de codes en bloc de paramètres $(n = 2 * k, k, d_{min})$ définis par des graphes construits avec de petits codes de base concaténés en série et en parallèle.
L'appellation "Cortex" vient de l'analogie de forme entre cette structure de codage et les neurones du néo-cortex d'un cerveau [24]. Les connexions entre les cellules du cerveau sont vaguement similaires aux connexions assurées par les permutations dans la structure.

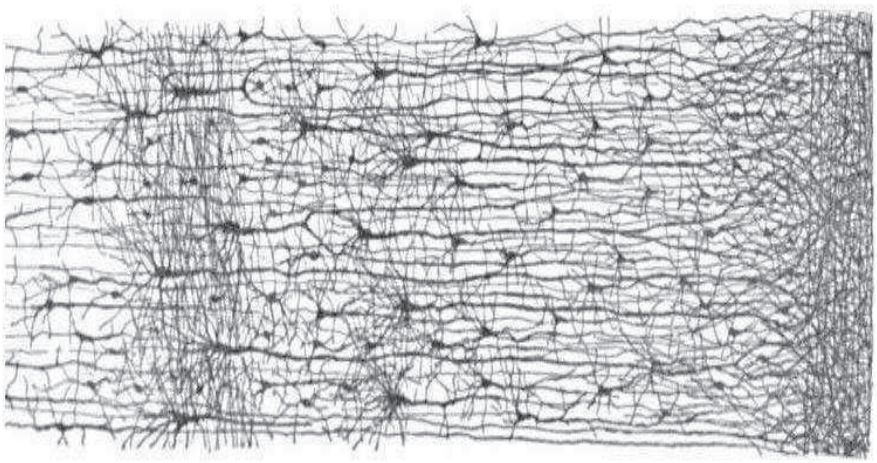

Figure 2.1. Dessin de neurones du nocortex.

La construction Cortex permet d'avoir des codes de petites longueurs n avec des distances minimales d_{min} très grandes.

L'étude du comportement du décodage itératif des codes Cortex a fait l'objet des travaux conduits par Olocco *et al* [68] peu après l'invention des codes Cortex. Dans cette étude, la structure Cortex est dépliée en graphe de Tanner dans lequel les noeuds de variables sont représentées par les variables d'entrées/sorties de chaque code de base et les noeuds de contrôle de parité sont représentés par les codes de base. Un des résultats de cette étude est que les performances de décodage itératif sont moins intéressantes si la structure Cortex comporte plus de 2 étages de permutations.

Les travaux ultérieurs d'Ayoub Otmani *et al* [69, 70] consistent à étudier les possibilités de construction des codes Cortex ayant des paramètres optimaux. Ces travaux ont aboutit à des résultats très intéressants qui montrent le potentiel de la structure Cortex pour construire des codes auto-duaux. Un de ces résultats montre que tous les codes de type II (c'est-à-dire que les poids de leurs mots de code sont multiples de 4) peuvent être obtenus sous forme des codes Cortex.

Le potentiel qu'offre la méthode Cortex pour construire des codes de grandes distances minimales est confronté à la difficulté posée par leur décodage.

Le décodage des codes Cortex est toujours un problème ouvert à cause des difficultés inhérentes à la structure Cortex et notamment à la présence des variables cachées dans cette structure.

La première tentative réussite de décodage des codes Cortex basée sur leur structure est introduite par Cadic *et al* dans [71]. Elle consiste à déplier la structure Cortex en une forme particulière de "bracelet" qui conduit à un treillis minimal décrivant le code et sur lequel un algorithme type Viterbi est effectué. Cette approche a permis de trouver des treillis minimaux pour certains codes comme le code Cortex-Golay$(24, 12, 8)$ décrit par un treillis à 16 états. Ce passage de la structure Cortex au treillis minimal décrivant le code ne peut pas être généralisé à toutes les structures Cortex car elle dépend en grande partie des permutations utilisées.

Dans [72, 73], Chamorro *et al* proposent un algorithme de décodage analogique qui consiste à constituer un graphe de Tanner à partir de la structure Cortex en remplaçant

les codes de base par leurs graphes de Tanner. Sur le graphe constitué, un algorithme de décodage type Belief-Propagation (BP) est effectué en propageant des messages entre ses différents noeuds en même temps et sans contrainte de l'horloge du système pour finalement calculer des estimations des LLR *a posteriori* des symboles. Cette approche est confrontée aux problèmes des cycles courts présents dans le graphe de Tanner des codes de base mais aussi dans le graphe global et qui dégradent les performances de l'algorithme BP.

Les travaux récents de Patrick Adde *et al* de Télécom Bretagne sur les codes Cortex ont aboutit à un algorithme de décodage à décision douce dit "à réencodage" [74]. C'est un algorithme de type Chase [76] qui génère une liste des mots candidats à partir des positions les moins fiables du mot reçu et en sélectionne ensuite le mot à distance euclidienne minimale du mot reçu. Les performances de cet algorithme sont très bonnes mais sa complexité devient très grande le rendant impraticable pour des codes de longueurs de plus de 48. Cet algorithme ne s'inspire pas vraiment de la structure Cortex comme les deux algorithmes présentés précédemment mais plutôt la propriété d'inversibilité des matrices génératrice et de contrôle du code.

Arzel *et al* de Télécom Bretagne proposent dans [77] un algorithme de décodage stochastique. Cet algorithme qui imite l'approche analogique proposée dans [72] en adoptant le même graphe de Tanner sur lequel des flux binaires aléatoire sont échangés entre les différents noeuds. Les performances obtenues pour le décodage du code Cortex$(32,16,8)$ sont d'environ 1dB moins bonnes que celle de l'algorithme proposé par Adde *et al* [74].

Dans le cadre de cette thèse, nous avons étudié la structure Cortex dans l'objectif de pouvoir mettre en oeuvre des algorithmes de décodage qui tirent avantage de la structure Cortex sans être contraints par la dimension et la distance minimale du code. Cependant, toutes nos tentatives ont été confrontées au problème de variables cachées et à la difficulté de leurs estimations à partir des symboles reçus. Mais comme disait Thomas Edison: *"Je n'ai pas échoué. J'ai simplement trouvé 10.000 solutions qui ne fonctionnent pas"*. Nous avons beaucoup appris de ces essais qui ne fonctionnent pas

et nous sommes persuadés que nous avons apporté une modeste contribution pour résoudre le problème de décodage des codes Cortex.

Dans la suite de ce chapitre, nous présentons la construction Cortex en exposant son principe général et en donnant quelques exemples de constructions. Puis, nous abordons le décodage des codes Cortex en présentant les meilleures méthodes de décodage déjà publiées dans la littérature avant de présenter notre contribution à ce sujet.

2.2 La construction Cortex

2.2.1 Principe de construction

La construction Cortex [51] utilise de petits codes de base C_b mis en parallèle dans un nombre d'étages en série séparés par des permutations, comme le montre la Figure 2.2. Les bits d'information $(d_0, d_1, \ldots, d_{k-1})$ présents à l'entrée de la structure de codage vont être traités par le premier étage des codes de base, les bits de sortie seront ensuite permutés avant d'être transmis aux codes de base de l'étage suivant. Le processus continue d'un étage à l'autre jusqu'au dernier étage qui délivre en sortie les bits de redondance $(r_0, r_1, \ldots, r_{k-1})$ formant le mot de code avec les bits d'information. Le mélange des bits entre étages, assuré par les permutations, permet d'augmenter la distance minimale du code et donc son pouvoir de correction. Ils existent beaucoup de permutations possibles ($k!$ permutations) entre 2 étages mais il est très difficile de trouver les permutations optimales permettant d'avoir des codes optimaux de distances minimales d_{min} maximales pour le code (n, k).

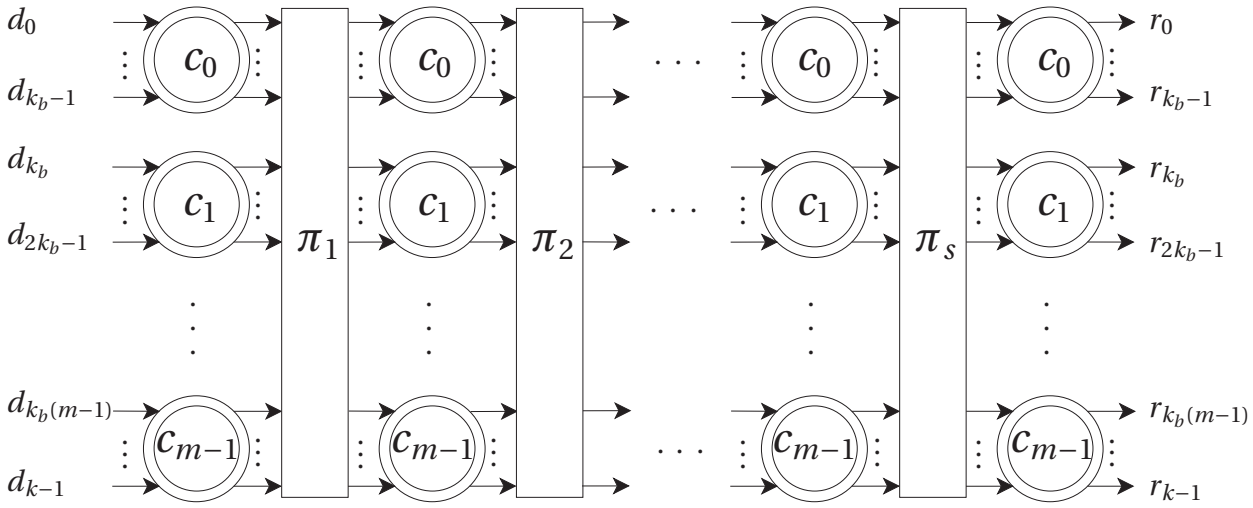

Figure 2.2. Schma gnral de la construction Cortex.

2.2.2 Exemples de construction Cortex

2.2.2.1 Construction du code de Hamming$(8,4,4)$ *avec code de base de Hadamard*$(4,2,2)$

Le code de Hamming$(8,4,4)$ peut être construit sous forme Cortex [24] avec 3 étages de 2 code de base Hadamard$(4,2,2)$, décrit par la Figure 2.3, et deux permutations comme le montre la Figure 2.4.

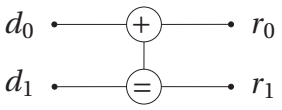

Figure 2.3. Graphe de Tanner du code de Hadamard$(4,2,2)$.

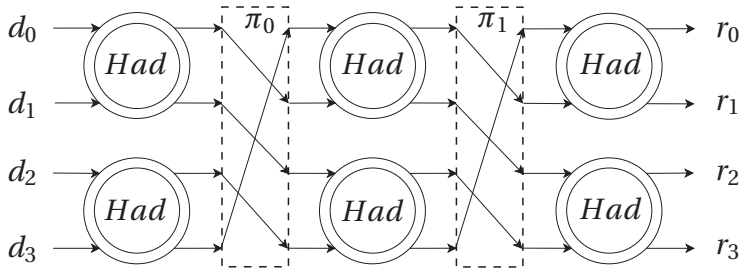

Figure 2.4. Code de Hamming$(8,4,4)$ sous forme Cortex.

où Had désigne le code de Hadamard$(4,2,2)$. La matrice de parité P associée à la structure cortex précédente peut être calculée comme suit:

$$P = P_e.P_{\pi_0}.P_e.P_{\pi_1}.P_e \tag{2.1}$$

où P_{π_0} (respectivement P_{π_1}) est une matrice de dimension $k \times k$ représentant la permutation π_0 (respectivement π_1) et P_e est la matrice représentant un étage. Ces matrices sont données par:

$$P_{\pi_0} = P_{\pi_1} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{2.2}$$

$$P_e = \begin{pmatrix} P_H & \mathbf{0} \\ \mathbf{0} & P_H \end{pmatrix}, \text{ où } P_H = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{2.3}$$

La matrice P est donc:

$$P = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix} \tag{2.4}$$

2.2.2.2 *Construction du code de Golay*$(24,12,8)$ *avec code de base de Hamming*$(8,4,4)$

Une structure Cortex du code de Golay$(24,12,8)$, présentée sur la Figure 2.5, est constituée de 3 étages. Chaque étage contient 3 codes de base de Hamming$(8,4,4)$. Les étages sont séparés par 2 permutations identiques π_0 et π_1 telles que: $\pi_0(i) = \pi_1(i) = 5 * i + 1$ modulo 12. Cette structure n'est pas unique mais elle à une caractéristique particulière lui permettant d'être dépliée sous forme d'un "bracelet"[71] conduisant ensuite à un treillis tail-biting à 16 états représentant le code de Golay$(24,12,8)$. Notre choix de présenter cette structure ici ne provient pas seulement de cette particularité mais aussi parce que tous les tentatives de décodage que nous allons présenter dans la suite, s'y sont basées.

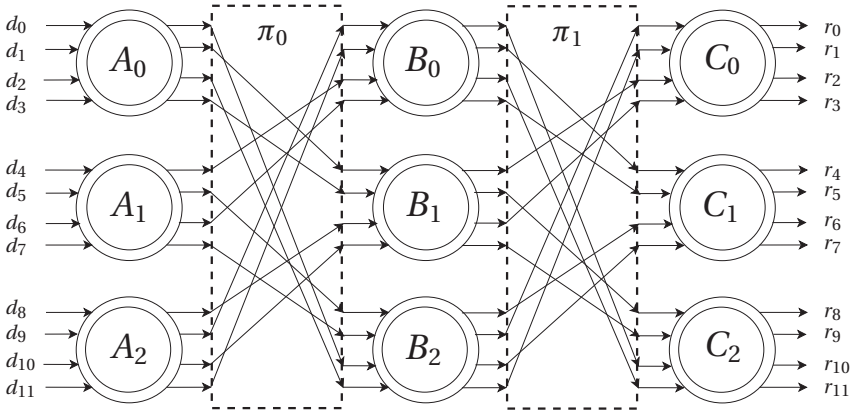

Figure 2.5. Code de Golay$(24,12,8)$ sous forme Cortex.

où A_i, B_i et C_i, $i = 0,1,2$, sont des codes de Hamming$(8,4,4)$.

La matrice génératrice du code de Golay(24,12,8) associée à la structure Cortex de la Figure 2.5 est donnée par la Figure 2.6. Sur cette figure, un cercle plein représente 1 et un cercle creux représente 0.

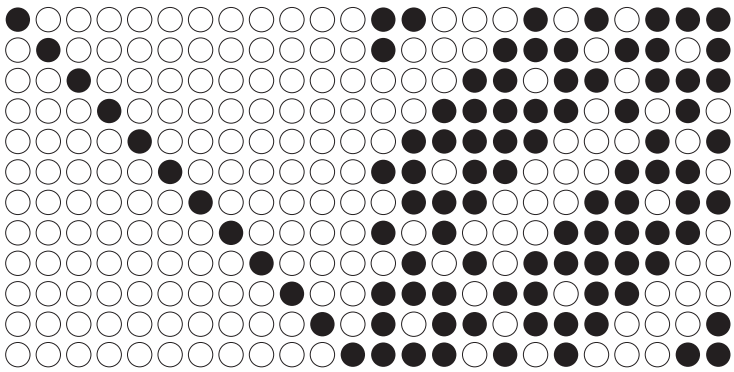

Figure 2.6. Matrice gnratrice du code de Golay$(24,12,8)$ associe la structure Cortex de la Figure 2.5.

2.2.2.3 Construction du code de $(48,24)$ *avec code de base de Hamming*$(8,4,4)$

La borne supérieure de la distance minimale des codes ($n = 48, k = 24$) est $d_{min} \leq 12$. La construction d'un code ($n = 48, k = 24, d_{min} = 12$) extrémal (dont la distance minimale est égale à la borne supérieure) sous forme Cortex avec 3 étages est toujours une question ouverte. Les travaux effectués jusqu'ici n'ont pas réussi à prouver l'existence ou la non-existence d'une telle construction Cortex. Une discussion dans [24] sur l'exitence d'une telle construction pose une question intéressante sur le lien entre le fait que le code $(48,24,12)$ ait une matrice de parité circulante et l'existence d'une construction Cortex de 3 étages. Cette question est motivée par les codes, de Hamming$(8,4,4)$ et de Golay$(24,12,8)$ dont les matrices de parité sont circulantes et représentés par des structures Cortex à 3 étages. Il est aisé de trouver un code $(48,24,8)$ avec 3 étages de 6 codes de Hamming$(8,4,4)$ comme le montre la Figure 2.7 [24]:

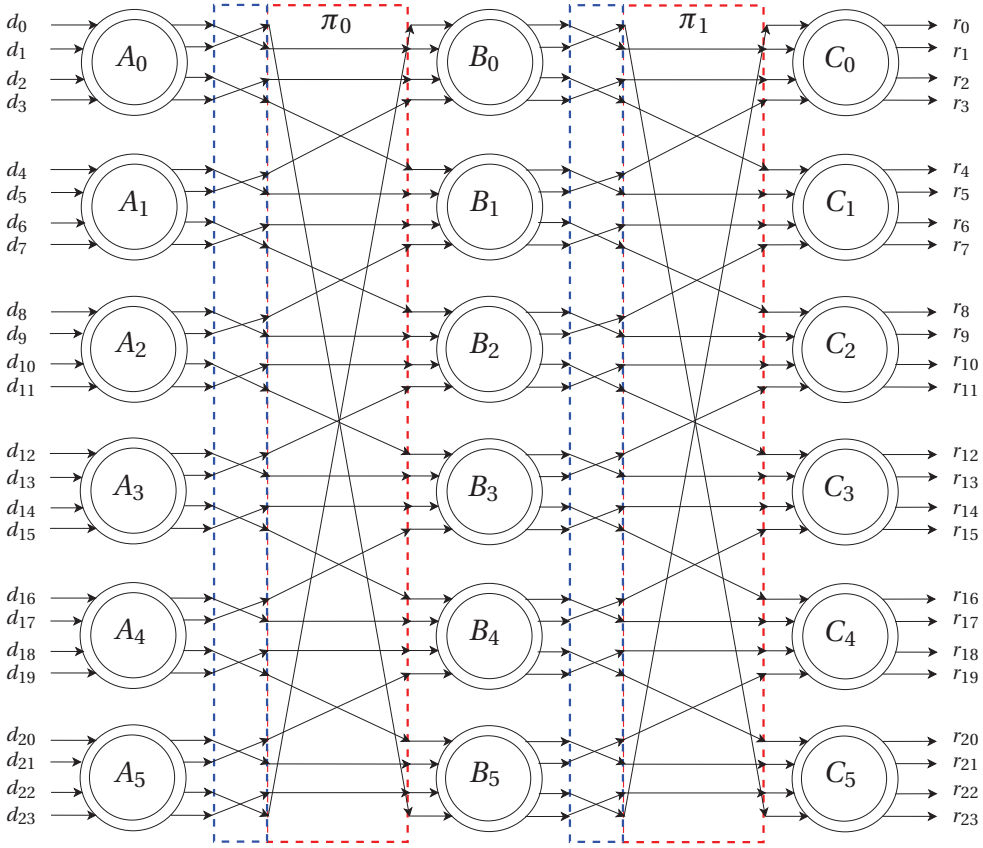

Figure 2.7. Code $(48,24,8)$ sous forme Cortex.

La matrice de parité P correspondante à la structure Cortex du code $(48,24,8)$ de la Figure 2.7 est donnée par:

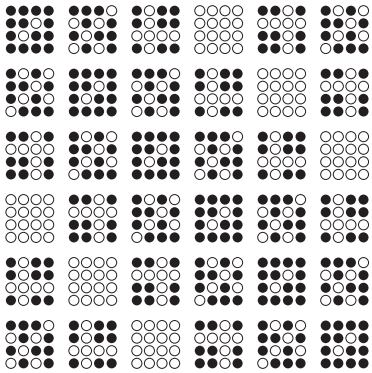

Figure 2.8. Matrice P du code $(48,24,8)$ de la Figure 2.7.

2.2.2.4 Construction du code de (72,36)

La construction Cortex d'un code extrémal (72,36,16) n'est pas encore trouvée [24]. Bien que la construction Cortex du code (72,36,12) ait été trouvée dans [24] en utilisant 5 étages de codes de Hamming(8,4,4) séparés par 4 permutations π_0, π_1, π_2 et π_3. Les permutations π_0 et π_1 résultent d'un partitionnement de la suite des entiers $\{0,1,\ldots,35\}$ en respectivement 9 et 4 vecteurs-lignes respectivement 4 et 9 éléments successifs pour former des matrices de dimensions respectivement 9×4 et 4×9, puis transposées et enfin mises à plat pour donner des vecteurs de 35 entiers permutéss telles que:

$$\pi_0 = \pi_3 = \{0,4,8,12,16,20,24,28,32,1,5,9,13,17,21,25,\ldots\}$$

$$\pi_1 = \pi_2 = \{0,9,18,27,1,10,19,28,2,11,20,29,3,12,21,30,\ldots\}$$

2.3 Décodage des codes Cortex

Dans cette section, nous présentons dans un premier temps deux algorithmes de décodage qui ont été proposés par des équipes de Télécom Bretagne pour le décodage des codes Cortex. En second temps, nous présentons les travaux que nous avons faits pour contribuer au décodage des codes Cortex en exploitant leurs structures.

Soit $\mathscr{C}(n,k)$ un code Cortex défini sur $\mathbb{F}_2$, $\boldsymbol{G} = [\boldsymbol{I}_k, \boldsymbol{P}]$ sa matrice génératrice dans sa forme canonique et $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ un mot de code de $\mathscr{C}$. La modulation utilisée est la modulation BPSK qui associe le bit 0 (respectivement 1) à la valeur modulée -1(respectivement $+1$). Le mot modulé $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ associé au mot $\boldsymbol{c}$ est ensuite transmis sur un canal à bruit blanc additif gaussien de variance $\sigma^2 = N_0/2$. Le mot reçu est noté $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$.

2.3.1 Décodage par liste

L'algorithme présenté ici [75] s'inspire de la méthode de décodage par liste proposée par Chase en 1972 [76]. Il génère 2 listes de vecteurs de test V_s et V_p à partir des positions des bits les moins fiables parmi les k bits d'information et les k bits de redondance respectivement. Les séquences de la liste V_s seront ensuite ré-encodées par la matrice $\boldsymbol{P}$ tandis que les séquences de la liste V_p seront ré-encodées par la matrice inverse de $\boldsymbol{P}$. Le décodeur sélectionne finalement le mot de code dans les 2 listes à distance euclidienne minimale du mot reçu. L'algorithme est adapté plutôt au décodage des codes dont la matrice $\boldsymbol{P}$ est inversible. C'est le cas pour les codes auto-duaux dont la matrice inverse de P est donnée par $\boldsymbol{P}^{-1} = \boldsymbol{P}^T$ où $\boldsymbol{P}^T$ désigne la matrice transposée de $\boldsymbol{P}$. Nous notons dans ce cadre que les codes Cortex qui sont construits à partir des codes de base auto-duaux héritent cette propriété d'auto-dualité [70].
Nous notons par $\boldsymbol{z} = (z_0, z_1, \ldots, z_{n-1})$ la décision dure associée au mot reçu $\boldsymbol{y}$ tels que: $z_i = 1$ si $y_i > 0$ et $z_i = 0$ si $y_i \leq 0$. Le vecteur $\boldsymbol{z}$ est ensuite séparé en 2 parties $\boldsymbol{z}_s$ et $\boldsymbol{z}_p$ correspondant, respectivement, aux k bits d'information et k bits de redondance. L'algorithme est résumé aux étapes suivantes [75]:

1. Génération de la liste des mots candidats à partir de $\boldsymbol{z}_s$

   1.1. Recherche des l_s bits les moins fiables de $\boldsymbol{y}_s$ où $\boldsymbol{y}_s$ est la partie de la séquence reçue correspondant à l'information.

   1.2. Construction d'une liste V_s de vecteurs de test à partir du vecteur $\boldsymbol{z}_s$ en inversant certains bits des l_s bits les moins fiables.

   1.3. Encodage des vecteurs de la liste V_s par la matrice $\boldsymbol{P}$ pour construire une liste des mots de code candidats.

   1.4. Calcul des distances euclidiennes entre les mots de code de la liste construit en 1.3 et le mot reçu $\boldsymbol{y}$.

2. Génération de la liste des mots candidats à partir de $\boldsymbol{z}_p$

   2.1. Recherche des l_p bits les moins fiables de $\boldsymbol{y}_p$ de la séquence reçue.

   2.2. Construction d'une liste V_p de vecteurs de test à partir du vecteur $\boldsymbol{z}_p$ en inversant certains bits des l_p bits les moins fiables.

   2.3. Encodage des vecteurs de la liste V_p par la matrice $\boldsymbol{P}^{-1}$ pour construire une liste des mots de code candidats.

   2.4. Calcul des distances euclidiennes entre les mots de code de la liste construit en 2.3 et le mot reçu $\boldsymbol{y}$.

3. Sélection du mot de code $\hat{\boldsymbol{c}}$ des 2 listes calculées en 1.4 et 2.4 à distance euclidienne minimale du mot reçu.

Le nombre et positions des l_s (respectivement l_p) bits les moins fiables à inverser dans la partie $\boldsymbol{z}_s$ (respectivement $\boldsymbol{z}_p$) changent d'un code en bloc à l'autre. Pour le décodage du code de Golay$(24,12,8)$, le nombre de bits les moins fiables à inverser dans les 2 parties information $\boldsymbol{z}_s$ et redondance $\boldsymbol{z}_p$ est $l_s = l_p = 8$. Les positions, numérotés de 1 à 8, des bits à inverser sont données par le Tableau 2.1 [78].

Bits à inverser dans z_s ou z_p	**Nombre de vecteurs de test**
Pas d'inversion	1
Le $i^{\text{ème}}$ bit, $i = 1,2,\ldots 8$, est inversé	8
Les bits $(1,i)$ sont inversés, $i = 2,\ldots 8$	7
Les bits $(2,i)$ sont inversés, $i = 3,\ldots 8$	6
Les bits $(3,4)$, $(3,5)$ sont inversés	2
Les bits $(1,2,i)$ sont inversés, $i = 3,4,5,6$	4
Les bits $(i,3,4)$, $(i,3,5)$, $i = 1,2$, sont inversés	4
Total de vecteurs de test	**32**

Table 2.1. Les positions des bits les moins fiables inverser pour le dcodage du code de Golay$(24,12,8)$

Les performances de l'algorithme de décodage pour le code de Golay$(24,12,8)$ obtenues avec les 32 vecteurs calculés dans la tableau précédent sont présentées sur la Figure 2.18.

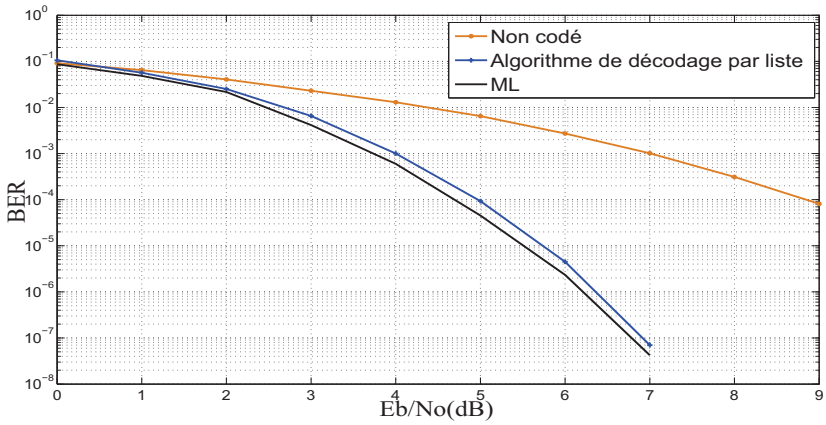

Figure 2.9. Performances de l'algorithme de dcodage par liste.

2.3.2 Décodage analogique des codes Cortex

Le décodage analogique des codes Cortex a été proposé dans [72]. Il consiste à utiliser la structure de codage Cortex pour établir un réseau analogique sur lequel un algorithme de décodage type Belief-Propagation (BP) est effectué en propageant des messages entre ses différents noeuds en même temps et sans contrainte de l'horloge du système (décodage analogique) pour finalement avoir des estimations des probabilités a posteriori des symboles. Le décodage commence par déplier la structure Cortex de base en remplaçant les codes de base par leurs graphes de Tanner. Sur ce nouveau graphe, dont les branches sont bidirectionnelles, un algorithme type Belief-Propagation est effectué. Les algorithmes type BP sont très sensibles aux courts cycles présents dans le graphe de Tanner du code. Pour augmenter la longueur du plus court cycle (appelé "girth") dans le graphe global du code Cortex, l'auteur de [72] utilise le code de Hadamard$(4,2,2)$ comme code de base dans les structures Cortex des différents codes à décoder. Le choix de ce code de base vient du fait que son graphe de Tanner (Figure 2.3) ne contient pas de cycles, ceci augmente le "girth" dans le graphe Cortex et donc le rend plus adapté à

un décodage type BP. Nous exposons ce type de décodage analogique à l'aide du code de Hamming$(8,4,4)$ dont la structure Cortex est présentée sur la Figure 2.4.
En remplaçant chaque code de base dans cette structure par son graphe de Tanner (Figure 2.3), le graphe Cortex du code de Hamming$(8,4,4)$ est donné par la Figure 2.10.

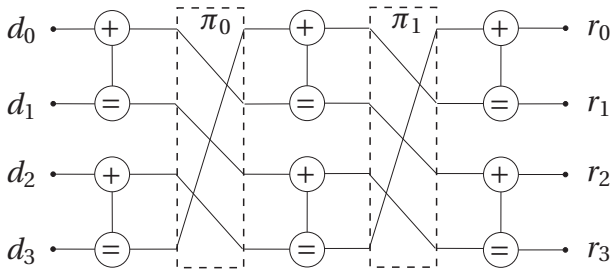

Figure 2.10. Graphe Cortex du code de Hamming$(8,4,4)$.

Le graphe Cortex est plus adapté au décodage type BP que le graphe de Tanner construit à partir de la matrice de contrôle du code car son girth est plus grand que celui du graphe de Tanner. La Figure 2.11 compare les performances de décodage analogique avec les performances de l'algorithme BP sur le graphe de Tanner et les performances du décodage optimal ML-exhaustif pour le code de Hamming(8,4,4). A un taux d'erreur binaire 10^{-5}, le décodage analogique apporte un gain d'environ 1 dB par rapport au décodage BP sur le graphe de Tanner du code et atteint les performances optimales du ML-exhaustif à 0.1 dB près.

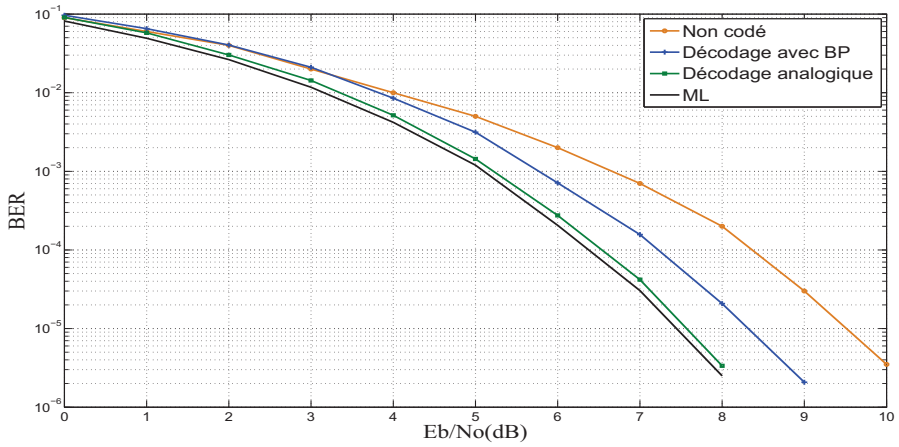

Figure 2.11. Performances du dcodage analogique pour le code de Hamming(8,4,4).

2.3.3 Décodage par inversion et propagation

Nous proposons ici un algorithme de décodage qui utilise la propriété des mots du code de Hamming$(8,4,4)$ qui fait que les 4 bits de redondance r_i (respectivement d'information d_i) se déduisent très simplement des 4 bits d'information d_i (respectivement de redondance r_i) par une inversion $v = d_0 \oplus d_1 \oplus d_2 \oplus d_3 = r_0 \oplus r_1 \oplus r_2 \oplus r_3$ telle que $r_i = d_i \oplus v$, $i = 0,1,2,3$. Cette propriété est exploitée pour estimer les métriques des variables internes à partir des symboles reçus. Nous nous référons à la structure Cortex du code de Golay$(24,12,8)$ présentée sur la Figure 2.5 pour exposer cet algorithme de décodage. Un décodage par propagation sur la structure nécessite une estimation des variables internes à la sortie du premier étage des codes de base (sortie de chacun des 3 codes de base de Hamming$(8,4,4)$) à partir des $k = 12$ symboles d'information reçus ainsi que celles à l'entrée du dernier étage de la structure (entrée de chacun des 3 codes de base de Hamming$(8,4,4)$) à partir des $k = 12$ symboles de redondance reçus.

2.3.3.1 *Estimation des métriques au niveau du code de base*

Dans la structure Cortex, nous ne disposons aucune information sur les variables internes car ces informations ne sont pas transmises. Nous utilisons la propriété d'inversion précédente pour estimer ces variables à partir des symboles du mot reçu. Un code de Hamming$(8,4,4)$ du premier étage (respectivement du dernier étage) de la structure Cortex de la Figure 2.5 qui reçoit 4 symboles d'information (respectivement 4 symboles de redondance) va ensuite estimer les métriques des 4 bits de redondance (respectivement d'information) en utilisant la propriété d'inversion. Par exemple, le premier code de base de Hamming$(8,4,4)$ du premier étage reçoit en entrée 4 symboles (y_0, y_1, y_2, y_3) du mot reçu. Pour estimer les métriques de ses 4 bits de sortie, il procède comme suit:

1. Les métriques $y_i^{(H)}$, $i = 0,1,2,3$, des 4 bits de sortie sont initialisées à 0. Un vecteur des métriques $\boldsymbol{y}^{(H)}$ est formé tel que $\boldsymbol{y}^{(H)} = (y_0, y_1, y_2, y_3, y_0^{(H)}, y_1^{(H)}, y_2^{(H)}, y_3^{(H)})$.
2. Le vecteur des métriques $\boldsymbol{y}^{(H)}$ est ensuite décodé par décodage ML-exhaustif sur les 16 mots du code de Hamming$(8,4,4)$. Soit $\boldsymbol{c}^{(H)}$ le mot de code obtenu.
3. Calcul de la valeur d'inversion $v = c_0^{(H)} \oplus c_1^{(H)} \oplus c_2^{(H)} \oplus c_3^{(H)}$.
4. Estimation des valeurs des métriques des 4 bits de sortie $y_i^{(H)}$, $i = 0,1,2,3$, telles que:

$$y_i^{(H)} = \begin{cases} |y_i| & \text{si } v = 0 \\ -|y_i| & \text{si } v = 1 \end{cases} \tag{2.5}$$

2.3.3.2 *Description de l'algorithme*

L'algorithme par inversion effectue un décodage sur la structure Cortex en faisant coopérer tous les codes de base qui la constituent. Les codes de base aux étages de bord de la structure reçoivent chacun une partie du mot reçu du canal alors que les codes de base à l'étage du milieu n'ont pas d'accès direct à ces informations. L'algorithme utilise les informations reçues pour faire une première estimation des variables internes à l'aide des

codes de base de bord avant de les passer aux codes de l'étage au milieu afin d'affiner ces estimations. L'algorithme fait donc des phases "aller-retour" entre les codes de base des étages de bord d'un côté et les codes de base de l'étage au milieu. L'algorithme est résumé aux étapes suivantes:

1. Dans la première phase, les codes de base du premier et dernier étage vont estimer les métriques des variables internes selon l'équation 2.5.
2. Les métriques des variables internes, estimés dans l'étape précédente, sont délivrés ensuite aux codes de base de l'étage au milieu. Chaque code de base de cet étage fait un décodage Max-Log-MAP et fournit des informations extrinsèques sur les 8 variables internes associées. Ces informations extrinsèques seront ensuite délivrées aux codes de base des étages de bord.
3. Chaque code de base des étages de bord qui vient de recevoir les informations extrinsèques sur ses 4 symboles internes, fait un algorithme Max-Log-MAP mais en utilisant les informations du canal sur les 4 autres symbole pour délivrer les LLRs des 4 symboles internes. A ce niveau chaque code de base des étage du bord donne une décision dure sur les 4 symboles du mot reçu pour constituer un mot global. Si ce mot représente un mot de code ou le nombre maximal d'itérations fixé est atteint, l'algorithme s'arrête. Sinon une nouvelle itération de l'algorithme commence à partir de l'étape 2.

2.3.3.3 Performances de l'algorithme

Dans cette section, nous présentons les courbes des taux d'erreur binaires ou BER (Bit Error Rate) obtenues avec le décodage du code de Golay$(24, 12, 8)$ (Figure 2.5) par l'algorithme par inversion décrit précédemment. Les simulations sont effectuées en considérant une transmission sur un canal AWGN avec une modulation à deux phases ou BPSK.

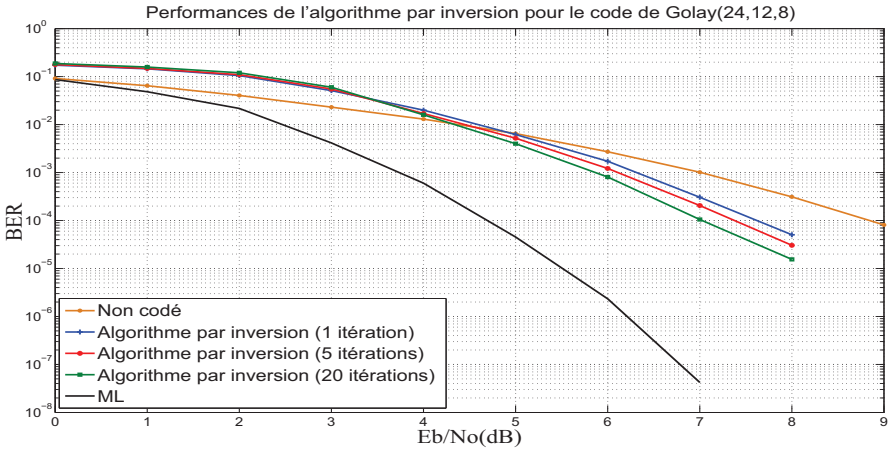

Figure 2.12. Performances de dcodage du code de Golay$(24, 12, 8)$ par l'algorithme par inversion.

Les courbes de simulations montrent que l'algorithme de décodage par inversion ne donne malheureusement pas d'excellents résultats. Nous pouvons en effet voir qu'il y a un gain en codage seulement à partir d'un rapport signal à bruit de l'ordre de 5 dB. A un faible rapport signal à bruit, il y a plus de symboles erronés dans le mot reçu et par conséquent plus d'erreurs sur les symboles estimés et transmis aux codes de l'étage central. Ces derniers n'arrivent pas à affiner ces estimations et font une sorte d'amplification de ces erreurs, ce qui cause une dégradation drastique des performances de décodage. Par contre à un fort rapport signal à bruit, il y a moins de symboles reçus erronés et donc moins d'erreurs transmises aux codes de base de l'étage central qui réussissent à mieux affiner les estimations et limiter la propagation rapide des erreurs. Cela a un effet positif sur les performances de décodage comme le montre les courbes des taux d'erreur binaire mais elles restent beaucoup plus mauvaises comparées à celles du décodage optimal ML-exhaustif. Les mauvaise performances de l'algorithme peuvent être expliquées par la présence des cycles dans la structure qui ramènent des informations aux codes de bases dont ils étaient origines et créent donc la corrélation qui dégrade les performances.

2.3.4 Décodage itératif par des formes booléennes

2.3.4.1 Equations booléennes obtenues au niveau de l'étage central

Pour décrire ces équations booléennes, nous rappelons qu'à la sortie de chaque code de Hamming$(8,4,4)$, chaque bit est une combinaison des trois bits d'entrée. Par exemple, le premier bit à la sortie du code A_0, de la figure 2.5, est égal à $d_1 \oplus d_2 \oplus d_3$. En d'autre termes, chaque bit à la sortie du premier étage est représenté par une équation booléenne des bits d'entrée. Ces équations vont, ensuite, être permutées par la permutation à gauche avant d'être mises à l'entrée des codes de base de l'étage central. Les mêmes démarches peuvent être aussi faites pour calculer les équations booléennes à la sortie des codes de base de cet étage central mais en allant des bits de redondance r_i et en utilisant le fait que la matrice de parité du code de Hamming$(8,4,4)$ est inversible et son inverse est lui même. Donc, la structure Cortex se réduit aux 3 codes de base de l'étage central avec des équations booléennes de part et d'autre. Nous appelons la partie composée d'un code de base B_i et ses équations booléennes associées la **forme booléenne** B_i. La figure 2.13 présente la nouvelle structure de codage avec 3 formes booléennes B_0, B_1 et B_2.

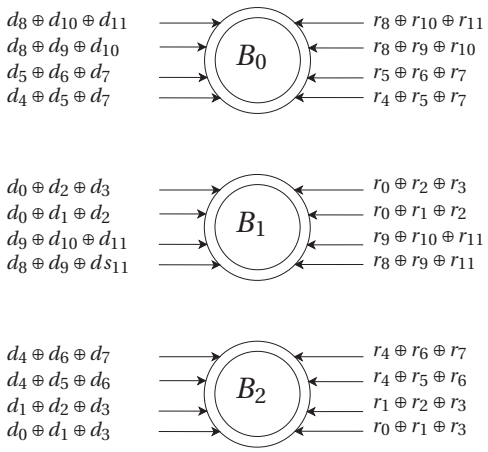

Figure 2.13. Equations boolennes obtenues au niveau de l'tage central

2.3.4.2 Structure en treillis de la forme booléenne de base

Chaque forme booléenne B_i, $i = 0,1,2$, de la Figure 2.13 est composée de trois parties, une première partie représentée par 4 équations booléennes impliquant 8 symboles d'information, une deuxième partie représentée par un code de Hamming$(8,4,4)$ et une troisième et dernière partie représentée par 4 équations booléennes impliquant 8 symboles de redondance. L'ensemble des équations booléennes de chaque partie peut être représenté par une matrice de 4 lignes et de 8 colonnes. Le nombre de lignes de la matrice est le nombre d'équations booléennes et le nombre de colonnes est déterminée par le nombre total des symboles (d'information ou de redondance) participant aux équations booléennes de la même partie. Par exemple, la partie gauche de la forme booléenne B_0 peut donc être décrite par la matrice $\boldsymbol{H}_0$:

$$\boldsymbol{H}_0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

Les 4 équations booléennes peuvent être calculée par le produit matriciel tel que:

$$(d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}).\boldsymbol{H}_0^T \tag{2.7}$$

Nous considérons le code en bloc $\mathscr{C}_0(8,4)$ dont une matrice de contrôle de parité est $\boldsymbol{H}_0$. Le code $\mathscr{C}_0$ peut être décrit par un treillis à 16 états obtenu à partir de la matrice $\boldsymbol{H}_0$ en utilisant la méthode de construction de Wolf-BCJR présentée dans le chapitre 1. Un tel treillis est décrit par la Figure 2.14. Par convention, une ligne solide représente un bit de code 0 alors qu'une ligne pointillée représente un bit de code 1.

Le treillis représentant la deuxième partie des équations booléennes de la forme booléenne B_0 est simplement l'image miroir du treillis décrit par la Figure 2.14.

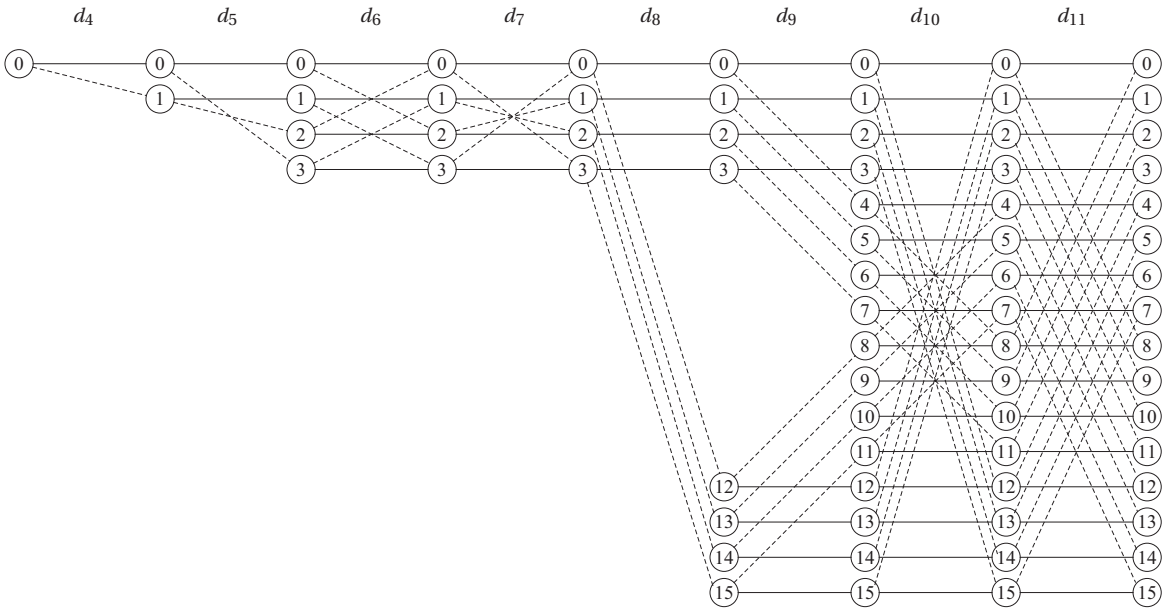

Figure 2.14. Treillis reprsentant la partie des quations boolennes gauche.

Le code de base B_i, $i = 0,1,2$, peut être vu comme une machine d'états qui fait des transitions entre un ensemble de 16 états d'entrée et un ensemble de 16 états de sortie. Chaque état d'entrée est représenté par les 4 symboles d'information et l'état d'arrivée correspondant est représenté par les 4 bits de redondance. La Figure 2.15 donne le diagramme d'états associé au code de base B_i.

Figure 2.15. Diagramme d'tats du code de Hamming$(8,4,4)$

Ce diagramme d'états peut être transformé en treillis à 16 états d'une seule section dont chaque état de départ est représenté par un vecteur de 4 bits d'entrée de ce code et l'état d'arrivée correspondant est représenté par les 4 bits de sortie. La Figure 2.16 décrit le treillis associé au code de base B_i, $i = 0,1,2$. Il peut être vu aussi comme une permutation de taille 16.

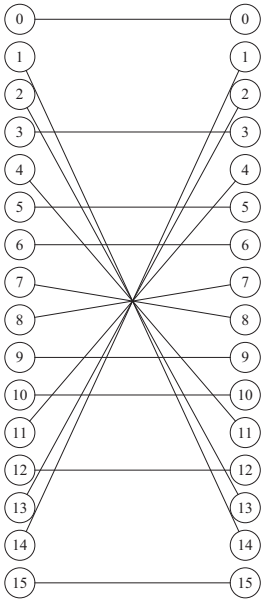

Figure 2.16. Treillis associ aux codes de base B_i.

Le treillis global représentant la forme booléenne B_0 de la Figure 2.13 est obtenu par la concaténation des 3 treillis représentant les 3 parties qui la constituent. Les treillis des 2 autres formes booléennes B_1 et B_2 sont obtenus de la même manière.

2.3.4.3 Décodage itératif par les formes booléennes

Entre 2 formes booléennes de la Figure 2.13, il y a 8 symboles qui sont communs. Le décodage itératif se fait donc entre les 3 décodeurs représentés par les 3 formes booléennes B_0, B_1 et B_2 en échangeant, suite à chaque itération, des informations extrinsèques sur les bits en communs. A chaque itération, 2 décodeurs échangent des informations extrinsèques sur les 8 bits en commun comme le montre la Figure 2.17.

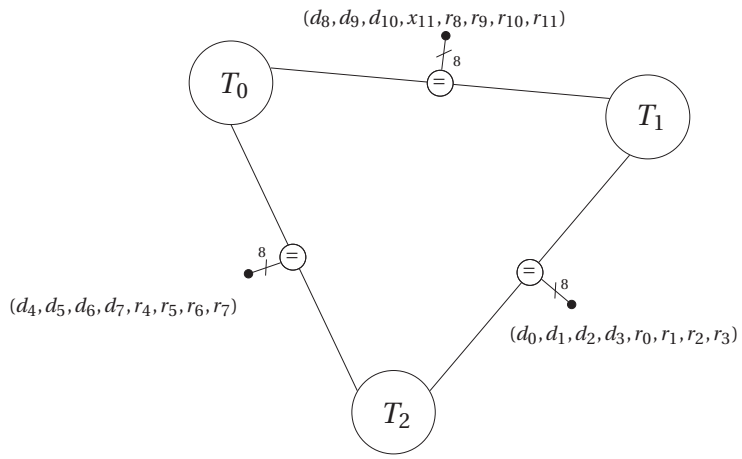

Figure 2.17. Structure de dcodage itratif de l'algorithme.

où T_i, $i = 0, 1, 2$, est le treillis global associé à la forme booléenne B_i.
L'algorithme est résumé aux étapes suivantes:

1. Effectuer un décodage SISO (Max-Log-MAP dans notre cas) sur chaque treillis T_i, $i = 0, 1, 2$ en utilisant les LLRs *a priori* des 16 symboles qu'il représente (8 symboles d'information + 8 symboles de redondance). Il délivre ensuite des informations extrinsèques sur ces 16 symboles.

2. Calcul des LLRs a posteriori des n bits du mot reçu tels que:

$$LLR(c_i) = LLR_{T_{j_1}}(c_i) + LLR_{T_{j_2}}(c_i). \tag{2.8}$$

   où $LLR_{T_{j_1}}$ et $LLR_{T_{j_2}}$ sont les deux LLRs fournis par les treillis T_{j_1} et T_{j_2} sur le bit commun c_i avec $j_1, j_2 \in \{0, 1, 2\}$.

3. Le mot $\hat{\boldsymbol{c}} = (c_0, c_1, \ldots, c_{n-1})$ correspondant aux LLRs précédents est ensuite déterminé tel que:

$$c_i = \begin{cases} 0 & \text{si } LLR(c_i) \leq 0 \\ 1 & \text{si } LLR(c_i) > 0 \end{cases} \tag{2.9}$$

   Si le mot $\hat{\boldsymbol{c}}$ est un mot de code ou le nombre maximal d'itérations fixé est atteint alors $\hat{\boldsymbol{c}}$ est déclaré comme décision sinon l'algorithme passe à l'étape suivante.

4. Echange des informations extrinsèques entre les 3 treillis sur les symboles en commun. A ce stade, chaque treillis T_i, $i = 0,1,2$, utilise les extrinsèques qu'il a reçues des autres treillis pour mettre à jour les LLRs *a priori* de ses 16 symboles.

5. Aller à l'étape 1.

2.3.4.4 Performances de l'algorithme

Dans cette section, nous présentons les courbes des taux d'erreur binaires ou BER (Bit Error Rate) obtenues avec le décodage du code de Golay$(24,12,8)$ (Figure 2.5) par l'algorithme par cet algorithme de décodage itératif. Les simulations sont effectuées en considérant une transmission sur un canal AWGN avec une modulation à deux phases ou BPSK.

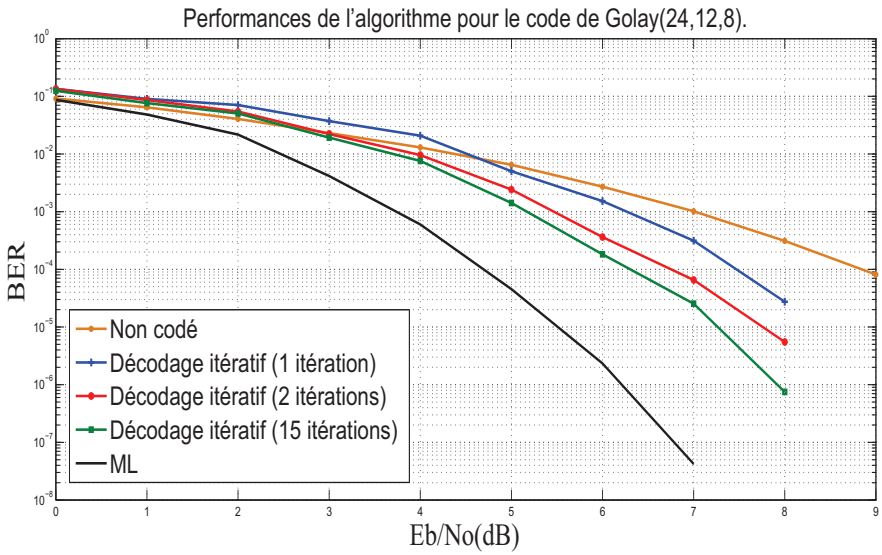

Figure 2.18. Performances de dcodage du code de Golay$(24,12,8)$ par l'algorithme des formes boolennes.

Les performances de l'algorithme de décodage par formes booléennes sont meilleures que celles de l'algorithme par inversion décrit dans le paragraphe précédent mais elles sont encore loin des performances optimales du ML. Cet algorithme de décodage ne nécessite pas d'estimation des variables internes car elles sont dissimulées dans les

équations booléennes qui se transforment en treillis. Par contre la forme en anneau de la structure de décodage de la Figure 2.17 pèse sur la convergence de l'algorithme. En effet, au bout de 3 itérations, les informations extrinsèques sur les symboles fournies à un des 3 treillis par les 2 autres treillis contiennent les informations *a priori* injectées à l'itération 1 car la structure de décodage est cyclique de cycle 3. Ce retour des informations *a priori* sur les endroits où elles y étaient créé de la corrélation et freine la vitesse de convergence de l'algorithme au delà de 3 itérations.

Certes, nous n'avons pas eu l'occasion d'aborder toutes les questions qui se posent autour de cet algorithme et les possibilités d'amélioration de ses performances. Nous posons ci-après certaines questions qui peuvent représenter des perspectives pour cet algorithme de décodage:

- Que se passe-t-il si l'étage central de la structure Cortex contient plus de 3 codes de base? Cela va certainement augmenter la longueur du cycle dans la structure de décodage, mais aura-il un impact en faveur des performances de l'algorithme?
- La structure en anneau est imposée par le fait que les treillis n'ont pas les même bits en commun. Si les treillis ont les mêmes bits en commun, la structure aura la forme d'un arbre et donc sans cycles. Cela améliorera-il les performances de décodage? Et que se passe-il si de plus tous les bits du mot de code sont représentés sur chaque treillis? Cela va certainement améliorer les performances car dans ce cas chaque bit est couvert par tous les treillis et donc profite davantage du décodage itératif mais les performances peuvent elles approcher les performances optimales?
- La taille des treillis (nombre d'états) est imposée par la dimension des codes de base. Dans le cas du code de Golay(24,12,8), les treillis sont à 16 états car la dimension du code de base est 4. Que se passe-il si les codes de base utilisés sont des codes de Hadamard$(4,2,2)$ conduisant ensuite à des treillis à 4 états? Ceci va certainement réduire la complexité globale de décodage.

2.4 Conclusion du chapitre 2

Dans ce chapitre, nous avons présenté une méthode appelée Cortex permettant de construire des codes en bloc linéaires à l'aide des petits codes en bloc mis en parallèle dans un nombre d'étages en série séparés par des permutations. Nous avons donné ensuite 4 exemples de construction des codes de Hamming$(8,4,4)$, Golay$(24,12,8)$, le code $(48,24,8)$ et le code $(72,32,12)$. L'avantage de la construction Cortex est qu'elle permet d'avoir une structure de codage factorisée de complexité réduite mais cet avantage ne peut pas, toujours, être exploitée dans le décodage à cause de la présence des variables internes dont les valeurs sont difficiles à estimer à partir des informations reçues. Nous avons présenté dans ce cadre certains travaux qui ont été faits pour mettre en oeuvre des algorithmes de décodage des codes Cortex.

Nous avons présenté dans un premier temps deux algorithmes de décodage qui ont été proposés par des équipes de Télécom Bretagne.

Le premier algorithme utilise les deux matrices génératrice et de contrôle du code pour générer une liste des mots candidats à partir de laquelle le mot le plus proche du mot reçu, en terme de distance euclidienne, est sélectionné.

Le deuxième algorithme est un décodage analogique qui consiste à déplier la structure Cortex en remplaçant les codes de base par leurs graphes de Tanner et faire ensuite un décodage type BP sur le graphe global pour finalement donner une estimation des probabilités *a posteriori* des symboles reçus.

Dans un deuxième temps, nous avons présenté les travaux que nous avons faits dans le cadre de cette thèse pour contribuer au décodage des codes Cortex. Nous avons présenté deux algorithmes basés sur la structure Cortex du code. Le premier algorithme utilise une propriété du code de base de Hamming$(8,4,4)$ pour estimer les métriques des variables internes à partir des symboles reçus et faire ensuite des phases "aller-retour" entre les étages de bord d'un coté et l'étage central dans lesquelles les codes de bases échangent des informations extrinsèques sur ces variables internes pour estimer

les LLRs *a posteriori* des symboles reçus. Le deuxième algorithme consiste à construire des treillis centrés autour des codes de base de l'étage central de la structure Cortex et les faire ensuite interagir dans un processus itératif en échangeant des informations extrinsèques sur les symboles reçus.

Ces deux algorithmes proposés ne donnent malheureusement pas d'excellents résultats mais ils représentent quand même une contribution, même si elle est modeste, au décodage des codes Cortex.

Chapitre 3

DÉCODAGE ITÉRATIF BASÉ SUR UN RÉSEAUX DE "PAPILLONS"

3.1 Introduction

Le décodage sur les treillis joue un rôle très important dans les systèmes de communications numériques. La représentation en treillis des codes en bloc conduit à la conception des algorithmes de décodage de performances optimales ou quasi-optimales avec une complexité réduite [45]. Le décodage sur treillis des codes en bloc trouve rapidement ses limites car la complexité de décodage dépend de celle du treillis qui croit exponentiellement avec la dimension du code.

Motivés par les opportunités qu'offrent les treillis, nous nous sommes investi dans le décodage des codes en bloc sur treillis tout en gardant en esprit que toute nouvelle tentative de décodage d'un code en bloc ne doit absolument pas passer par son treillis global mais elle doit permettre d'approcher les performances du décodage sur ce treillis global.

Avec ces objectifs ambitieux, nous avions un oeil ouvert sur les travaux de Kschischang *et al* [57] et Calderbank *et al* [41] sur les treillis des codes en bloc. Dans ces travaux, une méthode permettant de calculer le treillis global d'un code en bloc est présentée. Le treillis global d'un code en bloc est obtenu par le produit cartésien des plusieurs treillis élémentaires à 2 états issus des lignes de la matrice génératrice ou la matrice de contrôle du code.

Nous nous sommes rapidement posé la question sur les possibilités de faire un décodage d'un code en bloc en exploitant ces treillis élémentaires directement afin d'approcher les performances du décodage sur le treillis global avec une complexité réduite.

La méthode ci-avant [57][41] fait donc interagir les treillis élémentaires simultané-

ment à l'aide du produit cartésien pour donner le treillis global du code. Notre idée ensuite était de continuer dans cette direction c'est-à-dire faire interagir les treillis élémentaires mais d'une manière non simultanée en allongeant ce processus d'interaction sur plusieurs étapes où dans chaque étape, les treillis élémentaires s'interagissent 2 à 2. Nous souhaitions donc aboutir à une sorte de factorisation du treillis global.

L'idée de l'algorithme présenté ici est ensuite élaborée avec une nouvelle approche consistant à utiliser les probabilités d'états des treillis élémentaires pour concrétiser le résultat d'interaction entre ces deux treillis élémentaires.

Dans la suite nous donnerons une description formelle et détaillée de cet algorithme et nous étudierons ensuite ses performances ainsi que sa complexité.

Soit $\mathscr{C}(n,k,d_{min})$ un code en bloc linéaire défini sur $\mathbb{F}_2$ et $\boldsymbol{H}$ une matrice de contrôle du $\mathscr{C}$. Chaque ligne i de la matrice $\boldsymbol{H}$ est associé à un treillis élémentaire à 2 états T_i. Le produit entre tous les treillis élémentaires, $T_0 \otimes T_1 \otimes \ldots \otimes T_{n-k-1}$, conduit à un treillis à 2^{n-k} états représentant le code $\mathscr{C}$ [41]. Le produit $T_0 \otimes T_1 \otimes \ldots \otimes T_{n-k-1}$ assure donc une interaction simultanée entre tous les treillis élémentaires mais conduit à un treillis de très grande complexité. L'objectif de cette méthode de décodage est d'éviter le calcul du treillis global mais de faire interagir les treillis élémentaires 2 à 2 sur plusieurs étages afin d'approcher les performances du décodage sur le treillis global. Dans chaque étage, chaque treillis élémentaire T_i interagit avec un autre treillis T_j, $i \neq j$, à travers un opérateur que nous appelons "papillon". L'appellation "papillon" (en anglais butterfly) vient de l'analogie entre le réseau d'interactions des treillis élémentaires de l'algorithme proposé et le réseau d'opérateurs, appelés "butterfly", de l'algorithme de transformation de Fourier rapide ou FFT (Fast Fourier Transform) proposé dans [79]. L'opérateur "papillon" effectue le calcul des probabilités *a posteriori* des états sur le treillis-produit $T_i \otimes T_j$ des 2 treillis T_i et T_j selon l'algorithme BCJR[34] et par marginalisation des probabilités d'états de ce treillis-produit, il calcule les probabilités d'états de chacun des treillis élementaires T_i et T_j. Les treillis élémentaires T_i et T_j vont utiliser ces probabilités comme probabilités *a priori* d'états dans leurs interactions à

l'étage suivant. L'algorithme n'utilise donc pas en interne des probabilités extrinsèques sur les bits de code mais des probabilités *a posteriori* d'états.

Dans ce chapitre nous utiliserons le code de Hamming$(8,4,4)$ pour expliciter les différentes notions utilisées par l'algorithme. Le code de Hamming$(8,4,4)$ est défini par la matrice de contrôle suivante:

$$\boldsymbol{H}_{4\times 8} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.1}$$

Nous notons par T_0, T_1, T_2 et T_3 les treillis élémentaires associés respectivement aux lignes de la matrice $\mathbf{H_{4\times 8}}$.

Dans la suite du chapitre nous présenterons cette méthode de décodage à l'aide de la matrice de contrôle du code mais en réalité elle fonctionne aussi avec la matrice génératrice. Le choix entre les deux matrices est défini par le rendement $R = \frac{k}{n}$ du code. Si le rendement $R > \frac{1}{2}$, la matrice de contrôle représente un bon choix car dans ce cas elle contient moins de lignes que la matrice génératrice et donc conduit à une complexité de décodage plus réduite. Dans le cas contraire la matrice génératrice est plus adaptée et pour la même raison.

Dans ce chapitre nous allons tout d'abord décrire les notions nécessaires à la clarté de l'exposé de cet algorithme. La section 3.2 présente la notion d'un treillis élémentaire et donne quelques exemples simples permettant d'expliciter le calcul de ces treillis. La section 3.3 présente la notion du treillis-produit à l'aide de l'exemple de produit de deux treillis élémentaires représentant deux ligne de la matrice génératrice du code de Hamming(8,4,4). Le stockage des treillis-produits est indispensable pour le déroulement de l'algorithme. Cette section montre aussi qu'il suffit de stocker quatre sections possibles à partir desquelles le treillis-produit est facilement calculé. La section 3.4 décrit en détails l'algorithme et donne les étapes à suivre pour construire sa structure de

décodage. La section 3.5 présente les performances de l'algorithme pour le décodage de quelques codes en bloc. La section 3.6 étudie la complexité de l'algorithme. Finalement, une conclusion du chapitre est donnée dans la section 3.7.

Cet algorithme de décodage a fait l'objet d'un brevet [80].

3.2 Treillis élémentaire

Afin de pouvoir construire les treillis élémentaires, nous avons besoin de deux cellules notées $cell_0$ et $cell_1$ correspondant respectivement à un bit $h_{ij} = 0$ (respectivement $h_{ij} = 1$), $i = 0,1,\ldots,(n-k-1)$, $j = 0,1,\ldots,n-1$, de la matrice $\boldsymbol{H}$ du code. La Figure 3.1 présente les deux cellules $cell_0$ et $cell_1$.

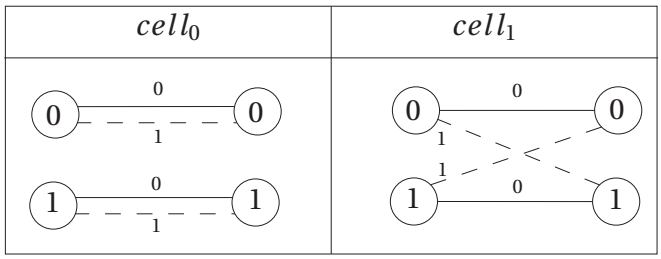

Figure 3.1. Structure des cellules $cell_0$ et $cell_1$

Un treillis élémentaire est donc constitué par la mise bout à bout des cellules $cell_0$ et $cell_1$ suivant la valeur des bits sur la ligne de la matrice représentant ce treillis. Par exemple les treillis élémentaire T_0 et T_1 représentant la première et deuxième ligne de la matrice de contrôle du code de Hamming$(8,4,4)$ sont présentés, respectivement, sur les Figure 3.2 et 3.3. Par convention, une ligne solide représente un bit de code 0 alors qu'une ligne pointillée représente un bit de code 1. Dans une section où ne figurent pas des lignes pointillées, une ligne solide représente deux branches distinctes. Ces conventions sont aussi valables dans tout ce chapitre.

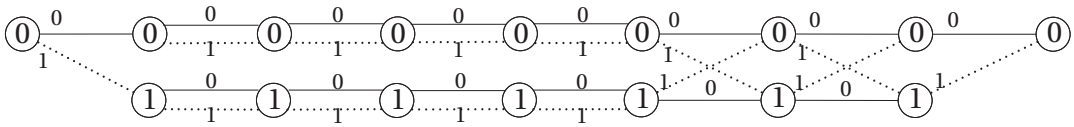

Figure 3.2. Treillis Imentaire T_0 associ la ligne $(1,0,0,0,0,1,1,1)$.

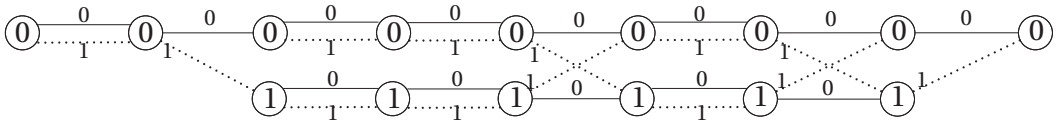

Figure 3.3. Treillis Imentaire T_1 associ la ligne $(0,1,0,0,1,0,1,1)$.

Dans le cas où la méthode de décodage utilise la matrice génératrice $\boldsymbol{G}$ du code, les treillis élémentaires sont construits d'une autre manière décrite déjà dans le paragraphe 1.2.6.2 du premier chapitre.

3.3 Treillis-produit

Une fois les treillis élémentaires sont calculés, l'algorithme effectue au niveau de chaque opérateur "papillon" le produit cartésien entre deux treillis élémentaires à 2 états chacun. Le produit effectué par un opérateur "papillon" dépend de la matrice utilisée dans le décodage. Dans le cas où la matrice génératrice est utilisée, le produit entre treillis est effectué de la même manière déja décrite dans le paragraphe 1.2.6.2 du premier chapitre. Dans le cas où la matrice de contrôle est utilisée, le produit cartésien entre deux treillis élémentaires est effectué de la manière suivante: considérons deux treillis $T^{(0)} = (V^{(0)}, E^{(0)}, \mathbb{F}_2)$ et $T^{(1)} = (V^{(1)}, E^{(1)}, \mathbb{F}_2)$, de même nombre n de sections d'indice $t \in \{0,1,\ldots,n-1\}$ tels que: $V^{(i)} = \{V_0^{(i)}, V_1^{(i)}, \ldots, V_n^{(i)}\}$ et $E^{(i)} = \{E_0^{(i)}, E_1^{(i)}, \ldots, E_{n-1}^{(i)}\}$ pour $i \in \{0,1\}$. Chaque section t d'un treillis $T^{(i)}$, $i \in \{0,1\}$, est un ensemble de branches d'étiquettes $e_t^i \in E_t^{(i)}$ connectant un état de départ $v_{t-1}^i \in V_{t-1}^{(i)}$ à un état d'arrivée $v_t^i \in V_t^{(i)}$.

Le produit cartésien entre $T^{(0)}$ et $T^{(1)}$ désigné par $T^{(0)} \otimes T^{(1)}$ est défini tel que [41]:

$$T^{(0)} \otimes T^{(1)} = \{((v_{t-1}^0, v_{t-1}^1), (e_t^0, e_t^1), (v_t^0, v_t^1))\} \text{ avec: } e_t^0 = e_t^1 \tag{3.2}$$

Le produit de deux treillis effectue donc le produit cartésien de chacune des trois composantes des triplet-branches mais en supprimant les branches si $e_t^0 \neq e_t^1$.
A titre d'exemple, le treillis résultant du produit cartésien entre les deux treillis élémentaires T_0 et T_1 est présenté sur la Figure 3.4.

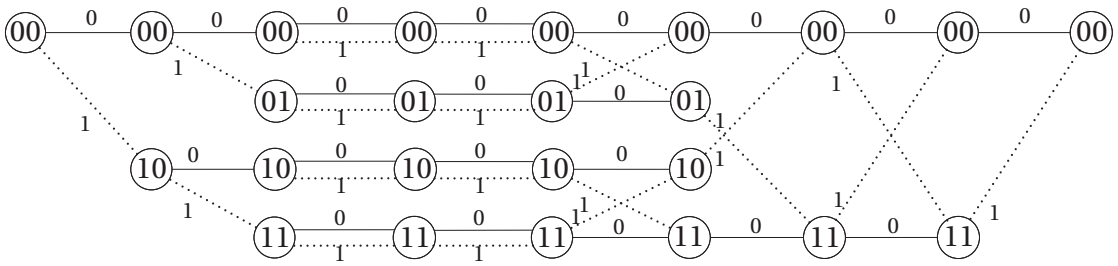

Figure 3.4. Treillis-produit $T_0 \otimes T_1$.

La section t dans le treillis-produit est obtenue par le produit cartésien entre la section t du treillis T_0 et la section t du treillis T_1. Cette section doit être un des 4 section-produits suivantes: $cell0 \otimes cell0$, $cell0 \otimes cell1$, $cell1 \otimes cell0$ ou $cell1 \otimes cell1$. Donc au lieu de stocker tous les treillis-produits, il suffit de stocker ces 4 sections et en faire la correspondance avec les bits (h_{im}, h_{jm}), des 2 lignes i et j, $i \neq j$, de la matrice de contrôle pour $i, j = 0, 1, \ldots (n-k-1)$ et $m = 0, 1, \ldots (n-1)$. Cela permet de réduire l'espace mémoire utilisé. Les 4 sections $cell0 \otimes cell0$, $cell0 \otimes cell1$, $cell1 \otimes cell0$ ou $cell1 \otimes cell1$ sont présentées respectivement sur les figures 3.5, 3.6, 3.7 et 3.8.

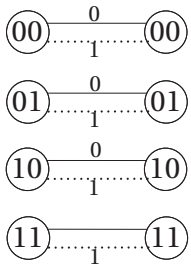

Figure 3.5. Section-produit $cell0 \otimes cell0$.

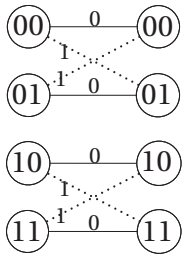

Figure 3.6. Section-produit $cell0 \otimes cell1$.

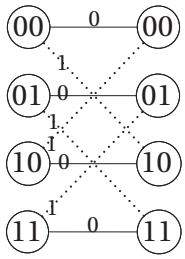

Figure 3.7. Section-produit $cell1 \otimes cell0$.

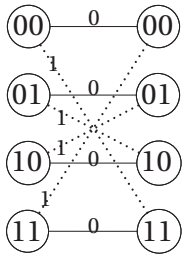

Figure 3.8. Section-produit $cell1 \otimes cell1$.

3.4 Description de l'algorithme

Le principe de l'algorithme de décodage est de faire interagir des treillis élémentaires 2 à 2 représentant les lignes de la matrice de contrôle (ou la matrice génératrice) du code en bloc sur plusieurs étages afin de calculer les probabilités *a posteriori* des bits reçus. Dans chaque étage, chaque treillis élémentaire T_i interagit avec un autre treillis T_j, $i \neq j$, à travers un opérateur "papillon". Tout treillis T_i est décrit par sa structure donnée par la ligne H_i de la matrice de contrôle qui lui est associée, et par l'ensemble des probabilités $\{Pr(s_t(T_i))\}$ de ses états. Ce sont ces données $\{Pr(s_t(T_i))\}$ qui sont transformées par les opérateur "papillon".

Avant de présenter les différentes étapes effectuées par l'algorithme en temps réel pour décoder le mot reçu $\boldsymbol{y}$, nous présentons tout d'abord les démarches à faire en amont pour déterminer la structure de décodage. La première démarche consiste à déterminer la matrice qui définit la carte d'interactions des treillis élémentaires via les opérateurs papillon. Nous décrivons dans le paragraphe suivant les règles à suivre permettant d'obtenir une telle matrice.

3.4.1 Réseau "papillon" et structure graphique de décodage

La matrice d'interactions des treillis élémentaires définit le réseau d'opérateurs "papillons" constituant la structure de décodage de la méthode. Le calcul de cette matrice suit les principes suivants:

- Tout opérateur "papillon" n'effectue des interactions qu'entre deux treillis T_i et T_j distincts, $i \neq j$.
- L'ensemble des opérateurs "papillon" d'un même étage couvre de façon bijective l'ensemble des treillis: *i.e.* les indices des treillis d'un étage forment une permutation de $\{0, 1, \ldots, (n-k-1)\}$.
- Globalement, tout couple de treillis (T_i, T_j), $i \neq j$, n'apparaît qu'une seule fois et

une seule dans toute la structure de décodage.

- Il existe au moins un chemin entre tout opérateur "papillon" du dernier étage et tout opérateur "papillon" du premier étage.

Pour le décodage d'un code en bloc linéaire $\mathscr{C}(n,k)$, la matrice d'interactions comporte $nEtage = \lceil Log_2(n-k) \rceil$ étages d'opérateurs "papillon" où $\lceil x \rceil$ désigne la partie entière supérieure de x.

Pour le décodage du code de Hamming$(8,4,4)$, la matrice d'interaction comporte $nEtage = \lceil Log_2(4) \rceil = 2$ étages d'opérateurs "papillon" et elle est donnée par le Tableau 3.1.

$$\begin{bmatrix} \text{étage 1} & \text{étage 2} \\ \downarrow & \downarrow \\ \begin{bmatrix} 0,1 \end{bmatrix} & \begin{bmatrix} 0,2 \end{bmatrix} \\ \\ \begin{bmatrix} 2,3 \end{bmatrix} & \begin{bmatrix} 1,3 \end{bmatrix} \end{bmatrix}$$

Table 3.1. Matrice des couples d'indices de treillis en interaction pour le code de Hamming(8,4,4).

où $[i,j]$, $i,j \in \{0,1,2,3\}$, sont les indices des lignes de la matrice $\boldsymbol{H}_{4\times 8}$ participant à un opératuer papillon.

A l'instar de la matrice d'interactions, la structure graphique de décodage du code de Hamming$(8,4,4)$ est également composée de deux étages. Le premier étage réalise les interactions entre les couples des treillis élémentaires (T_0,T_1) et (T_2,T_3) et le deuxième étage réalise les interactions entre les couples (T_0,T_2) et (T_1,T_3), ce qui peut se représenter graphiquement comme dans la Figure 3.9.

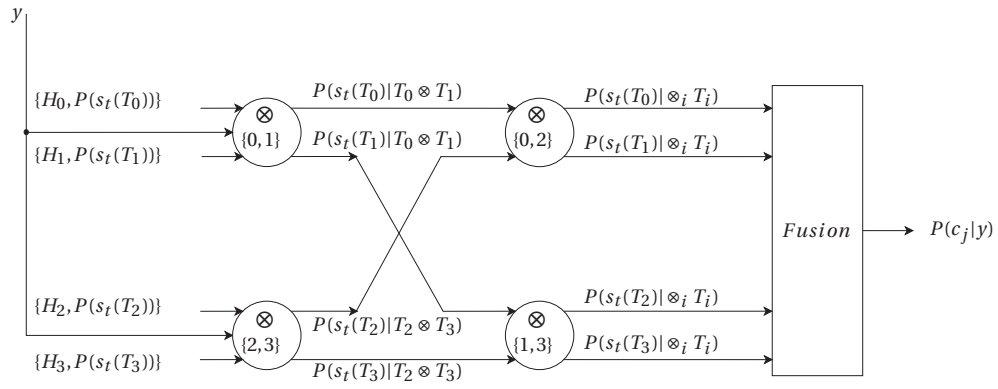

Figure 3.9. Structure globale de l'algorithme de dcodage pour le code de Hamming$(8,4,4)$.

Où $\otimes_i T_i = T_0 \otimes T_1 \otimes T_2 \otimes T_3$.

Le mot reçu $\boldsymbol{y}$ du canal est délivré seulement aux opérateurs "papillon" du premier étage de la structure afin d'éviter les corrélations qui peuvent se produire quand il est utilisé par les "papillon" de l'étage suivant. Les autres "papillons" reçoivent les données des treillis élémentaires avec leurs probabilités *a posteriori* calculées à l'étage précédent. Les calculs effectués par un opérateur "papillon" sont détaillés dans le paragraphe suivant.

3.4.2 Calculs effectués par un opérateur "papillon"

Un opérateur "papillon" effectue un algorithme d'estimation de probablilités *a posteriori* des bits d'états sur le treillis-produit $T_i \otimes T_j$ de deux treillis élémentaires T_i et T_j, $i \neq j$ selon l'algorithme de type BCJR. Il reçoit en entrée le mot reçu $\boldsymbol{y}$ (s'il est au premier étage), les 2 treillis élémentaires T_i et T_j et les probabilités de leurs états $\{P(s_t(T_i))\}_t$ et $\{P(s_t(T_j))\}_t$ respectivement. Puis il procède comme suit:

1. Calculer le treillis-produit $T_i \otimes T_j$ de deux treillis élémentaires T_i et T_j. Avec la donnée des lignes H_i et H_j de la matrice $\boldsymbol{H}$, la structure du treillis-produit $T_i \otimes T_j$ est

facilement obtenue à partir des 4 section-produits déjà stockées (voir section 3.3). Les probabilités *a priori* d'états de ce treillis-produit sont ensuite calculées à l'aide des probabilités *a priori* d'états $\{P(s_t(T_i))\}_t$ et $\{P(s_t(T_j))\}_t$ des treillis élémentaires T_i et T_j respectivement telles que:

$$P(s_t(T_i \otimes T_j) = (a,b)) = P(s_t(T_i) = a) \times P(s_t(T_j) = b) \tag{3.3}$$

2. Effectuer l'algorithme BCJR sur le treillis-produit $T_i \otimes T_j$ en utilisant les probabilités d'états $P(s_t(T_i \otimes T_j))$ calculées dans l'étape 1 comme probabilités *a priori* d'états. Une fois les deux phases "forward" et "backward" de l'algorithme BCJR sont effectuées sur ce treillis, le "papillon" calcule les probabilités *a posteriori* d'états du treillis-produit. Pour un état s_t de l'étage t du treillis $T_i \otimes T_j$, sa probabilité *a posteriori* est donnée par:

$$P(s_t(T_i \otimes T_j)) = \alpha_t(s_t) \times \beta_t(s_t) \tag{3.4}$$

où $\alpha_t(s_t)$ et $\beta_t(s_t)$ sont, respectivement, les probabilités "forward" et "backward" calculées par l'algorithme BCJR à l'état s_t (voir le paragraphe 1.4.3 du premier chapitre pour détails sur les probabilités $\alpha_t(s_t)$ et $\beta_t(s_t)$).

3. Calculer les probabilités *a posteriori* d'états des treillis T_i et T_j par marginalisation des probabilités d'états $P(s_t(T_i \otimes T_j))$ du treillis-produit $T_i \otimes T_j$.

$$P(s_t(T_i) | T_i \otimes T_j = a) = \sum_b P(s_t(T_i \otimes T_j) = (a,b)) \tag{3.5}$$

$$P(s_t(T_j) | T_i \otimes T_j = b) = \sum_a P(s_t(T_i \otimes T_j) = (a,b)) \tag{3.6}$$

Le mot reçu est utilisé seulement par les opérateurs "papillon" du premier étage de la structure de décodage. Les opérateurs "papillon" aux étages suivants n'exploitent pas directement ce mot reçu afin d'éviter la corrélation. Ils reçoivent seulement les

probabilités d'états des treillis élémentaires qu'ils manipulent. A cet égard, nous pouvons représenter graphiquement un opérateur "papillon" suivant qu'il se situe au premier étage ou non. La Figure 3.10 représente un opérateur "papillon" au premier étage alors qu'un opérateur "papillon" aux étages suivants est représenté par la Figure 3.11.

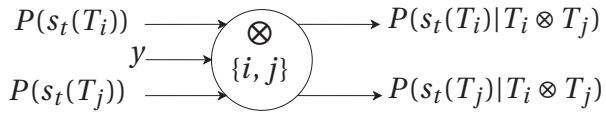

Figure 3.10. Schma reprsentatif d'un oprateur papillon au premier tage.

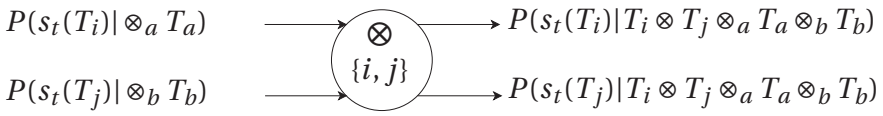

Figure 3.11. Schma reprsentatif d'un oprateur papillon ne se situant pas au premier tage.

Où les ensembles notés $\otimes_a T_a$ (respectivement $\otimes_b T_b$) représentent les treillis ayant déjà interagis avec le treilli élémentaire T_i (respectivement T_j) dans les étages précédents de la structure de décodage.

3.4.3 Description formelle de l'algorithme

Dans le paragraphe 3.4.1, nous avons présenté les démarches permettant de déterminer la structure de décodage pour un code en bloc.

Dans cette section, nous donnons une description plus détaillée des différentes étapes de décodage effectuées dans cette structure. Nous exposons ces étapes à l'aide de la structure de décodage simple du code de Hamming(8,4,4) décrite par la Figure 3.9.

Le décodage sur cette structure commence de l'étage gauche des opérateurs "papillon". Chaque "papillon" de cet étage reçoit le mot reçu $\boldsymbol{y}$ et les structures des 2 treillis élémentaires T_i et T_j, $i, j = 0, 1, 2, 3$, représentés, respectivement, par les deux lignes H_i

et H_j de la matrice de contrôle et dont les probabilités d'états $\{P(s_t(T_i))\}_t$ et $\{P(s_t(T_j))\}_t$ sont initialisées toutes à 0.5. Il effectue ensuite les calculs des probabilités *a posteriori* $P(s_t(T_i)|T_i \otimes T_j)$ $P(s_t(T_j)|T_i \otimes T_j)$ des états des treillis élémentaires T_i et T_j respectivement en suivant les démarches décrites dans le paragraphe 3.4.2 précédent.

Les structures des treillis élémentaires T_m, $m = 0,1,2,3$, avec leurs probabilités *a posteriori*, qui viennent d'être calculées par les papillons du premier étage, seront ensuite délivrées aux "papillons" du deuxième étage. Ces derniers procèderont ensuite de la même manière que ceux du premier étage en utilisant les probabilités *a posteriori* d'états comme des probabilités *a priori* pour calculer des nouvelles probabilités *a posteriori* d'états $P(s_t(T_m)|T_0 \otimes T_1 \otimes T_2 \otimes T_3)$, $m = 0,1,2,3$, de chaque treillis élémentaire T_m.

Les treillis élémentaires T_m, $m = 0,1,2,3$ avec leurs probabilités *a posteriori* $P(s_t(T_m)|T_0 \otimes T_1 \otimes T_2 \otimes T_3)$ seront réinjectés à l'entrée de la structure (sans le mot reçu $\boldsymbol{y}$) si une nouvelle itération de l'algorithme est prévue. Dans le cas où le nombre maximal d'itérations est atteint, l'algorithme entame sa phase finale.

Dans la phase finale (bloc $Fusion$ dans la Figure 3.9), l'algorithme de décodage calcule les probabilités *a posteriori* des bits de code c_j, $j = 0,1,\ldots,7$ en fusionnant les données fournies par chaque treillis élémentaire. Ce bloc multiplie toutes les probabilités finales d'un même bit de code de chacun des treillis élémentaires T_i pour $i = 0,1,2,3$ à partir de toutes les probabilités *a posteriori* des états de tous les 4 treillis élémentaires de l'étage final $P(s_t(T_i) \mid T_0 \otimes T_1 \otimes T_2 \otimes T_3)$. Cette phase fusionne donc toutes les informations disponibles sur chaque bit c_j, $j = 0,1,\ldots,7$, sur les 4 treillis élémentaires. La probabilité *a posteriori* d'un bit c_j est donnée par:

$$P(c_j \mid \mathbf{y}) \approx \prod_{i=0}^{4} P(c_j \mid T_i, \mathbf{y}) \tag{3.7}$$

où $P(c_j \mid T_i, \mathbf{y})$ est la probabilité *a posteriori* du bit c_j, $j = 0,1,\ldots,7$, calculée par le treillis élémentaire T_i, $i = 0,1,2,3$.

3.4.4 Décodage du code de Golay(24,12,8)

La structure globale de l'algorithme de décodage pour le code de Golay de paramètres ($n = 24, k = 12, d_{min} = 8$) consiste d'abord à trouver la matrice d'interactions correspondante composée de $nEtage = \lceil Log_2(n-k) \rceil$ permutations de l'ensemble d'indices $\{0,1,\ldots,n-k-1\}$. Nous avons donc $nEtage = 4$, et un ensemble de 4 permutations possibles comme décrit le Tableau 3.2.

$$\begin{bmatrix} \text{étage 1} & \text{étage 2} & \text{étage 3} & \text{étage 4} \\ \downarrow & \downarrow & \downarrow & \downarrow \\ \left[\,0,1\,\right] & \left[\,0,2\,\right] & \left[\,0,4\,\right] & \left[\,1,6\,\right] \\ \left[\,2,3\,\right] & \left[\,4,6\,\right] & \left[\,8,1\,\right] & \left[\,11,4\,\right] \\ \left[\,4,5\,\right] & \left[\,8,10\,\right] & \left[\,5,9\,\right] & \left[\,9,2\,\right] \\ \left[\,6,7\,\right] & \left[\,1,3\,\right] & \left[\,2,6\,\right] & \left[\,7,0\,\right] \\ \left[\,8,9\,\right] & \left[\,5,7\,\right] & \left[\,10,3\,\right] & \left[\,5,10\,\right] \\ \left[\,10,11\,\right] & \left[\,9,11\,\right] & \left[\,7,11\,\right] & \left[\,3,8\,\right] \end{bmatrix}$$

Table 3.2. Matrice des couples d'indices de treillis en interaction pour pour le code Golay$(24,12,8)$

où $\left[i,j\right]$, $i,j \in \{0,1,2,\ldots,11\}$, sont les indices des lignes de la matrice du contôle du code de Golay(24,12,8) participant à un opératuer papillon.

La structure globale de décodage pour le code de Golay(24, 12, 8) peut donc se représenter comme le montre la Figure 3.12.

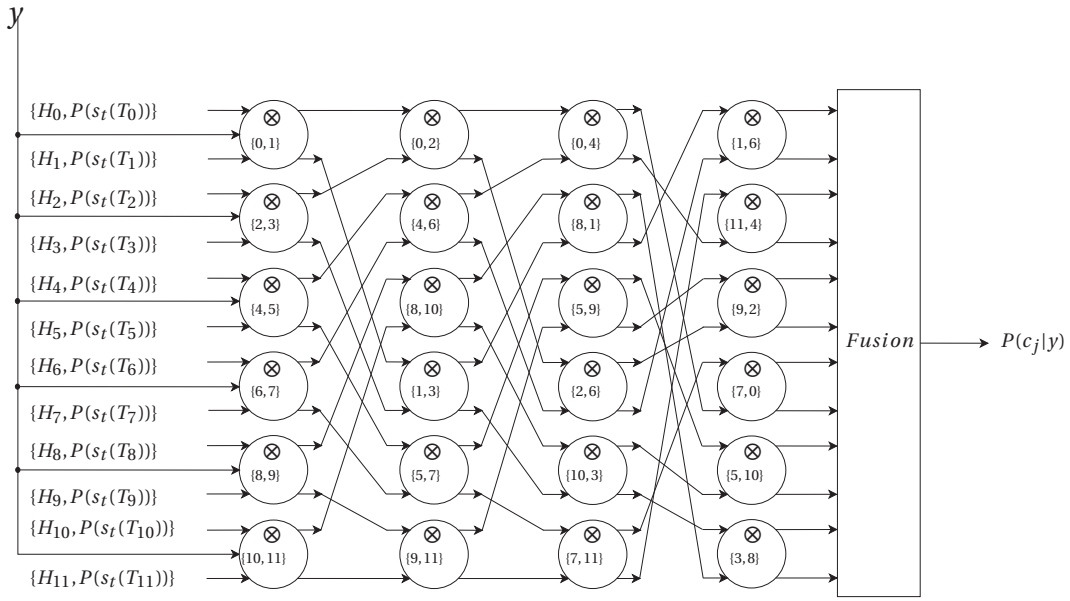

Figure 3.12. Structure globale de l'algorithme de dcodage pour le code de Golay$(24,12,8)$.

3.5 Performances de l'algorithme

Dans ce paragraphe, nous présentons les résultats de simulations obtenus avec cet algorithme pour le décodage des 3 codes de Hamming$(8,4,4)$, de Golay$(24,12,8)$ et QR$(48,24,12)$. Les Figure 3.13, 3.14 et 3.16 présentent, respectivement, les performances de l'algorithme pour ces 3 codes sur un canal binaire à effacement ou BEC. Dans ce cas de canal, le symbole reçu y_i, $i = 0,1\ldots,n-1$, est donné tel que $y_i = c_i$ si le bit n'a pas été effacé et $y_i = 0.5$ s'il est effacé. Pour le nombre d'effacements inférieur ou égal à $d_{min}-1$, nous avons fait un test exhaustif alors que pour un nombre d'effacements supérieur à $d_{min}-1$, les effacements sont introduits aléatoirement sur le mot de code. Nous constatons que la méthode corrige parfaitement tous les patterns d'au plus $d_{min}-1$ effacements pour les deux codes de Hamming$(8,4,4)$ et de Golay$(24,12,8)$ et elle corrige aussi certaines patterns de plus de $d_{min}-1$ effacements. Pour le code QR$(48,24,12)$, les performances ne sont malheureusement pas bonnes et la méthode n'arrive pas à atteindre la limite de

$d_{min} - 1$ effacements corrigés. Nous donnerons à la fin de cette section une éventuelle justification de ces dégradations de performances constatées pour le décodage de ce code.

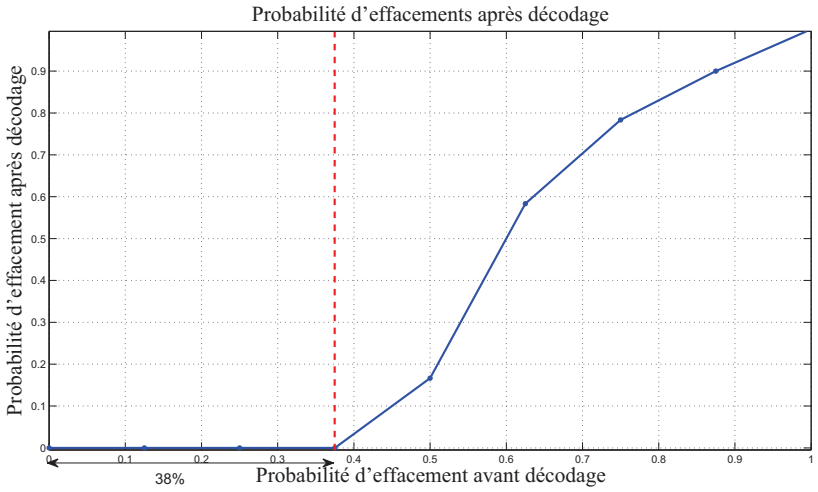

Figure 3.13. Performance de l'algorithme sur un canal BEC pour le code de Hamming$(8,4,4)$.

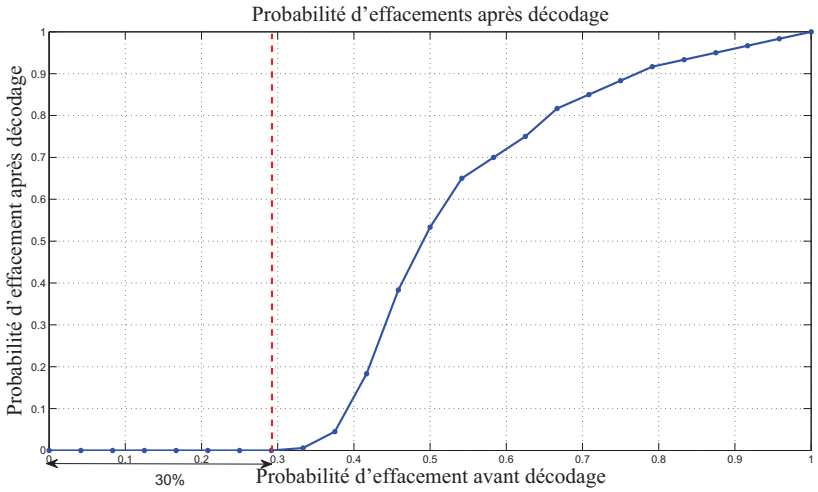

Figure 3.14. Performance de l'algorithme sur un canal BEC pour le code de Golay$(24,12,8)$.

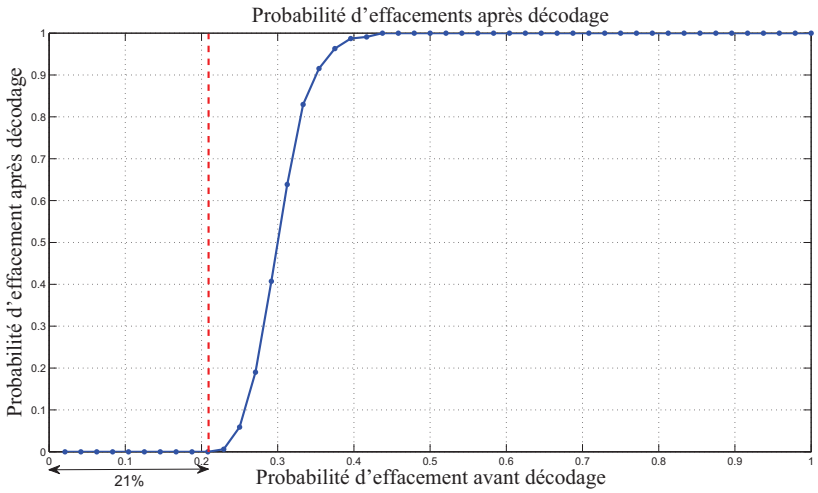

Figure 3.15. Performance de l'algorithme sur un canal BEC pour le code QR$(48,24,12)$.

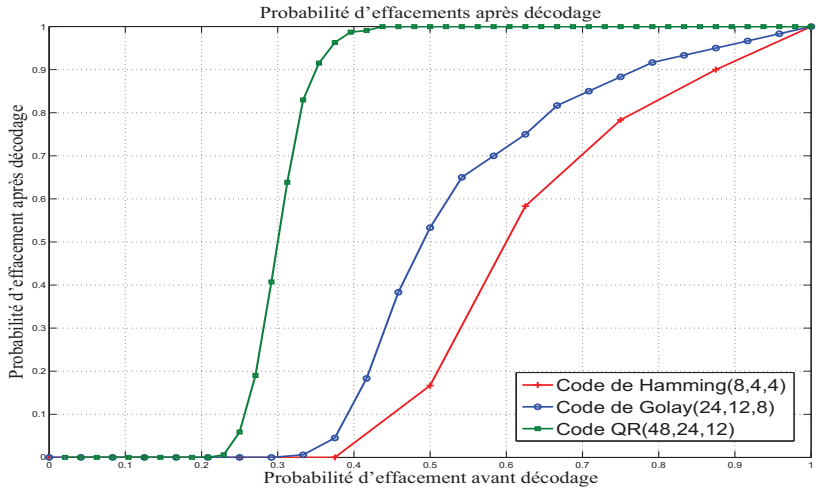

Figure 3.16. Comparaison des performances de l'algorithme sur un canal BEC pour les codes prcdents.

La Figure 3.17 présente les performances de l'algorithme pour le code de Hamming$(8,4,4)$ sur un canal perturbé par un bruit blanc additif gaussien ou AWGN. Dans ce cas, le symbole reçu y_i est la somme de la valeur modulée de c_i avec un bruit blanc additif gaussien de variance $\sigma^2 = N_0/2$ où N_0 est la densité spectrale de puissance du bruit.

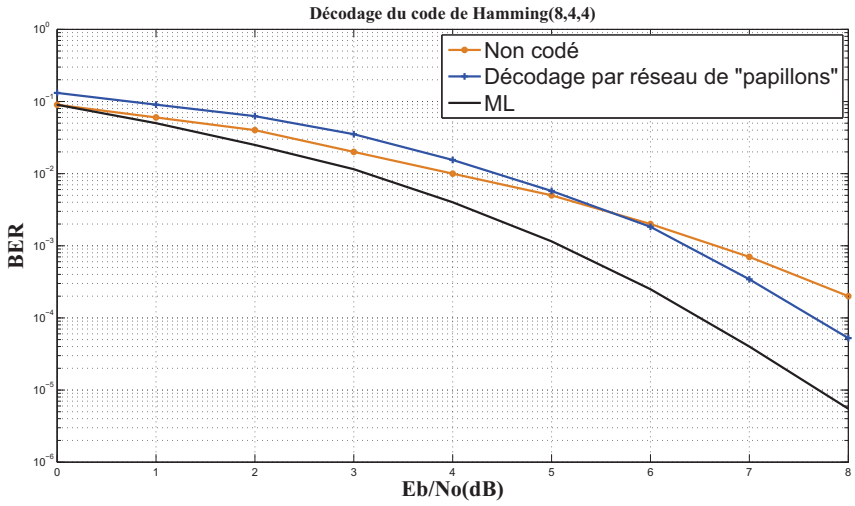

Figure 3.17. Performance de l'algorithme sur un canal AWGN pour le code de Hamming$(8,4,4)$.

Les courbes de simulations présentées sur la Figure 3.17 montrent que l'algorithme de décodage par réseau de "papillons" ne donne malheureusement pas de bons résultats sur un canal AWGN. Nous pouvons en effet voir qu'il y a un gain en codage seulement à partir d'un rapport signal à bruit de plus de 6 dB.

Nous pensons que les mauvaises performances de la méthode sont induites par la présence des sections nulles sur les treillis-produits. Une section nulle S_t est une section où chaque états s de l'étage t est connecté à un état homologue s de l'étage $t+1$ par l'ensemble des branches possibles. Deux états de cette section ne sont pas connectés entre eux s'ils sont différents. A titre d'exemple, les sections 3 et 4 du treillis décrit par la Figure 3.4 sont des sections nulles. Pour le code de Golay(24,12,8), les treillis-produits utilisés dans le décodage contiennent chacun 10/24 sections nulles! L'inconvénient d'une section nulle est que les symboles reçus correspondants ne contribuent pas aux calculs des probabilités *a posteriori* d'états des treillis. En effet, la présence des symboles correctement reçus sur des sections nulles d'un treillis-produit augmente le pourcentage des symboles erronés participant aux calculs des probabilités d'états conduisant ensuite à des valeurs des probabilités d'états moins per-

tinentes. Ces dernières vont ensuite affecter les probabilités d'états des autres treillis élémentaires dans les interactions suivantes, ce qui conduit à une sorte d'amplification et de propagation des erreurs. Certes, ces symboles présents sur ces sections nulles peuvent aussi être reçus erronés et donc réduisent le pourcentage des symboles erronés mais le problème se pose sur d'autres treillis-produits dans lesquels ces symboles couvrent des sections non-nulles.

Certes, nous n'avons pas confirmé ce problème par des moyens pratiques mais nous donnons ci-après deux propositions qui peuvent représenter des perspectives pour, peut être, y remédier dans le sens où elles permettront de réduire le nombre de sections nulles dans les treillis-produits.

- La première proposition consiste à faire une sectionnalisation des treillis-produits en fusionnant les sections nulles avec d'autres sections non-nulles pour former une seule section. La sectionnalisation permet également de réduire la complexité de décodage. Cette notion de sectionnalisation des treillis est présentée en détails dans le dernier chapitre de ce manuscrit.

- La deuxième proposition consiste à augmenter le nombre de treillis élémentaires formant un treillis-produit. Dans notre cas, les treillis-produits sont formés par le produit cartésien de 2 treillis élémentaires. Nous pouvons utiliser plus de 2 treillis élémentaires (par exemple 4) car cela réduit le nombre de sections nulles dans le treillis résultant et éventuellement limiter leurs effets négatifs sur les performances.

3.6 Etude de la complexité

La complexité de cet algorithme est, principalement, la complexité d'un opérateur "papillon" multipliée par le nombre total des ces opérateurs dans la structure de décodage. Un opérateur "papillon" effectue un algorithme BCJR sur le treillis-produit $T = T_i \otimes T_j$ de 2 treillis élémentaires T_i et T_j à 2 états pris sur l'ensemble des k treillis

élémentaires. Pour étudier la complexité d'un "papillon", nous allons calculer sa complexité pour une section t du treillis-produit T et ensuite faire la somme sur l'ensemble des sections constituant le treillis-produit. Ceci permet aussi de donner une étude générique qui peut être utilisée dans le cas où le papillon traite plus de 2 treillis élémentaires comme nous l'avons proposé dans les perspectives.

Considérons donc une section t du treillis-produit T, composée de n_s^{t-1} états de départ, n_s^t états d'arrivée et un nombre total de n_b^t branches. Cette section est le résultat du produit entre la section t du treillis T_i et la section t du treillis T_j. Un papillon qui reçoit les probabilités d'états des 2 treillis élémentaires T_i et T_j va tout d'abord calculer les probabilités d'états du treillis-produit résultant en utilisant l'équation 3.3, ceci demande pour la section t du treillis T, n_s^t multiplications. Le papillon effectue, ensuite, un algorithme BCJR sur le treillis T pour calculer les probabilités d'états forward α_t et backward β_t. Le calcul des probabilités α_t, à l'aide de équation 1.47, demande $N_a^t(\alpha_t)$ additions et $N_m^t(\alpha_t)$ multiplications où $N_a^t(\alpha_t)$ et $N_m^t(\alpha_t)$ sont donnés, respectivement, par:

$$N_a^t(\alpha_t) = n_b^t - n_s^t \tag{3.8}$$

$$N_m^t(\alpha_t) = n_b^t \tag{3.9}$$

Le calcul des probabilités β_t à l'aide de équation 1.48, demande $N_a^t(\beta_t)$ additions et $N_m^t(\beta_t)$ multiplications où $N_a^t(\beta_t)$ et $N_m^t(\beta_t)$ sont donnés, respectivement, par:

$$N_a^t(\beta_t) = n_b^t - n_s^{t-1} \tag{3.10}$$

$$N_m^t(\beta_t) = n_b^t \tag{3.11}$$

Après les calculs des probabilités α_t et β_t, l'algorithme calcule les nouvelles probabilités d'états du treillis-produit en utilisant l'équation 3.4, ceci demande n_s^t multiplications. Dans la dernière étape, l'opérateur "papillon" calcule les probabilités d'états de chacun des treillis élémentaires T_i et T_j par marginalisation des probabilités d'états du treillis-

produit T, ceci demande n_s^t additions. Donc la complexité totale d'un opérateur "papillon" est de $N_m^t(\otimes)$ multiplications et $N_a^t(\otimes)$ additions telles que:

$$N_a(\otimes) = \sum_{t=0}^{n-1} (N_a^t(\alpha_t) + N_a^t(\beta_t) + n_s^t) \tag{3.12}$$

$$N_m(\otimes) = \sum_{t=0}^{n-1} (N_m^t(\alpha_t) + N_m^t(\beta_t) + 2n_s^t) \tag{3.13}$$

Pour donner un ordre de grandeur de la complexité de la méthode, nous la calculons pour le code de Golay($n = 24, k = 12, d_{min} = 8$). Pour $k = 12$ et $n = 24$, la structure de décodage est donc composée de $nEtage = \lceil Log_2(n-k) \rceil = 4$ étages et chaque étage comporte $(n-k)/2 = 6$ papillons. Un opérateur "papillon" traite des treillis-produits binaires à 4 états. Donc, une section t est composée, en moyenne, de $n_s^{t-1} = 4$ états de départ, de $n_s^t = 4$ états d'arrivée et $n_b^t = 8$ branches. La complexité d'un opérateur "papillon" peut être réduite en exploitant la présence des sections nulles dans les treillis élémentaires et ensuite dans les treillis-produits. Dans une section nulle, les probabilités d'états de départ et celles d'états d'arrivée sont égales, cela veut dire qu'il suffit de calculer les probabilités d'états d'un étage pour en déduire les probabilités d'états de l'étage suivant. Dans un treillis-produit à 4 états, il y a 10 sections nulles. Cela veut dire qu'il y a 10 sections dans le treillis-produit où on n'a pas besoin d'y faire le calcul des probabilités d'états, ce qui peut se traduire par une compression du treillis à 14 sections seulement.

Donc, la complexité réelle d'un opérateur "papillon" est de 168 additions et 336 multiplications.

La complexité totale de la méthode de décodage pour le code de Golay$(24, 12, 8)$ est donc $4 \times 6 \times 168 = 4\,032$ additions et $4 \times 6 \times 336 = 8\,064$ multiplications.

3.7 Conclusion du chapitre 3

Dans ce chapitre nous avons présenté un algorithme de décodage pour les codes en bloc basé sur un réseau d'opérateurs permettant de faire interagir des treillis élémentaires représentant les lignes de la matrice génératrice ou de contrôle du code en bloc. Les opérateurs sont rangés dans plusieurs étages formant ainsi la structure globale de l'algorithme de décodage. Chaque opérateur du réseau fait interagir deux treillis élémentaires T_i et T_j, distincts $i \neq j$, via leur produit cartésien, effectue ensuite un algorithme BCJR sur le treillis-produit et par marginalisation des probabilités d'états de ce dernier calcule les probabilités *a posteriori* d'états de chacun des deux treillis élémentaires qui seront utilisées dans les étages suivants de la structure. L'algorithme se termine par le calcul de la probabilité *a posteriori* des bits reçus. Nous avons ensuite présenté les performances de la méthode en donnant les résultats de simulations pour les codes de Hamming$(8,4,4)$, de Golay$(24,12,8)$ et de QR$(48,24,12)$ dans le cas d'un canal à effacement ou BEC. L'algorithme corrige parfaitement tous les patterns d'au plus $d_{min}-1$ effacements pour les deux codes de Hamming$(8,4,4)$ et de Golay$(24,12,8)$ et elle corrige aussi certaines patterns de plus de $d_{min}-1$ effacements. Pour le code QR$(48,24,12)$, les performances ne sont malheureusement pas bonnes et la méthode n'arrive pas à atteindre la limite de $d_{min}-1$ effacements corrigés.

Nous avons clôturé ce chapitre par une étude détaillée de la complexité de la méthode en donnant le nombre d'opérations arithmétiques effectuées et nous avons donné un ordre de grandeur de la complexité en la calculant pour le code de Golay$(24,12,8)$.

Chapitre 4

DÉCODAGE DES CODES EN BLOC BASÉ SUR DES TREILLIS-PRODUITS

4.1 Introduction

Dans ce chapitre nous allons présenter deux algorithmes de décodage itératifs pour les codes en bloc basés sur des treillis-produits. Ces deux algorithmes utilisent des treillis-produits de complexité réduite construits à partir des treillis élémentaires représentant des lignes de la matrice de contrôle du code.

Le premier algorithme est un algorithme qui fait une recherche du mot le plus probable sur chacun des treillis-produits au lieu de faire la recherche sur le treillis global du code. Il utilise un décodage par List-Viterbi [81] sur chacun des treillis-produits.

Le deuxième algorithme est un algorithme SISO qui estime les logarithmes des rapports de vraisemblance ou LLRs (Log-Likelihood Ratio) *a posteriori* des symboles reçus. Cet algorithme utilise des techniques de décodage classiques sur treillis comme l'algorithme BCJR[34].

Pour la clarté de l'exposé de ces deux algorithmes, nous définissons tout d'abord quelques notations qui seront utilisées tout au long de ce chapitre.
Soit $\mathscr{C}(n,k)$ un code en bloc linéaire défini sur $\mathbb{F}_2$ et $c = (c_0, c_1, \ldots, c_{n-1})$ un mot de code de $\mathscr{C}$. Nous considérons une modulation à deux phases ou BPSK (Binary Phase Shift Keying) qui associe le bit 0 (respectivement 1) à la valeur modulée -1(respectivement $+1$). Le mot modulé $x = (x_0, x_1, \ldots, x_{n-1})$ associé au mot c est ensuite transmis sur un canal à bruit blanc additif gaussien de variance $\sigma^2 = N_0/2$ et à fading d'amplitude a. Le mot reçu est noté $y = (y_0, y_1, \ldots, y_{n-1})$ tel que $y_i = a.x_i + b_i$ où b_i est l'échantillon du bruit gaussien.

Soit $\boldsymbol{H}$ une matrice de contrôle du code $\mathscr{C}$. Nous notons par n_L, le nombre de lignes de $\boldsymbol{H}$ formant un treillis-produit, et par n_p le nombre de treillis-produits issus de la matrice $\boldsymbol{H}$. Les treillis-produits issus des lignes de la matrice $\boldsymbol{H}$ sont notés T_{p_i}, $i = 0, 1, \ldots, n_p - 1$ et ils sont donnés par:

$$T_{p_i} = T_{i*n_L} \otimes T_{i*n_L+1} \otimes \ldots \otimes T_{i*n_L+(n_L-1)} \tag{4.1}$$

Où T_m, $m \in \{0, 1, \ldots, n-k-1\}$, représente le treillis élémentaire associé à la m$^{\text{ème}}$ ligne de la matrice $\boldsymbol{H}$ et $\otimes$ designe le produit cartésien des treillis tel qu'il est présenté dans le chapitre 3.

4.2 Décodage itératif dérivé de l'algorithme List-Viterbi

Le concept du décodage itératif utilisé dans cette méthode de décodage est différent du concept classique basé sur l'échange des informations extrinsèques entre les décodeurs de base. Le concept du décodage itératif utilisé dans cet algorithme vient simplement du fait que le décodage est effectué en plusieurs étapes.

Cette technique utilise, à l'étape i de son processus, l'algorithme de décodage List-Viterbi [81] pour rechercher, sur chaque treillis-produit, le $i^{\text{ème}}$, $i = 1, 2, \ldots$, chemin le plus probable pour constituer une liste des mots candidats. L'algorithme sélectionne ensuite, dans cette liste, le mot de code à distance euclidienne minimale du mot reçu comme décision.

Dans ce paragraphe nous allons décrire cet algorithme en détaillant les différents calculs effectués, à chaque étape de décodage, par les décodeurs élémentaires représentés par les treillis-produit.

Mais nous allons tout d'abord présenter l'algorithme de décodage List-Viterbi [81].

4.2.1 Algorithme de décodage List-Viterbi

Un algorithme de décodage List-Viterbi est un algorithme basé sur l'algorithme de Viterbi et qui fait une recherche d'une liste de i chemins les plus probables dans un treillis

décrivant le code [81]. Ils existent deux variantes de cet algorithme. La première variante PLVA (Parallel List-Viterbi Algorithm) fait une recherche parallèle et fournit simultanément la liste des i chemins les plus probables. La deuxième variante SLVA (Serial List-Viterbi Algorithm) fait une recherche en série qui génère le $i^{\text{ème}}$ chemin le plus probable sachant les $(i-1)$ les plus probables déjà déterminés. Nous utilisons cette dernière variante SLVA car elle est mieux adaptée à l'algorithme itératif proposé.

Dans ce paragraphe nous allons tout d'abord présenter le principe de décodage SLVA en présentant les différents calculs qu'il effectue sur un treillis pour déterminer la liste des i chemins les plus probables. Pour cela, nous introduisons tout d'abord quelques notations nécessaires à la clarté de l'exposé de cet algorithme.

Soit T un treillis de n sections décrivant un code en bloc $\mathscr{C}(n,k)$. Nous notons par s_j^t, $(j=0,1\ldots)$, l'état j de l'étage t du treillis T, par $\hat{\Gamma}_{s_j^t}$ et $\bar{\Gamma}_{s_j^t}$, respectivement, les deux métriques des chemins survivant et concurrent à l'état s_j^t, par $\hat{s}_j^t$ l'état du chemin survivant à l'étage $(t-1)$ du treillis T, par $\bar{s}_j^t$ l'état du chemin concurrent à l'étage $(t-1)$ du treillis T et par $p_i=(p_i(0),p_i(1),\ldots,p_i(n))$ le $i^{\text{ème}}$ chemin le plus probable, où $p_i(t)$, $t=0,1,\ldots,n$, est l'état de p_i situé à l'étage t du treillis T.

La Figure 4.2.1 décrit un exemple d'un treillis de 5 sections sur lequel sont illustrées toutes les notations précédentes. Sur ce treillis, le chemin tout à zéro est considéré comme le chemin survivant à l'état s_0^t et l'autre chemin en trait gras est le chemin concurrent.

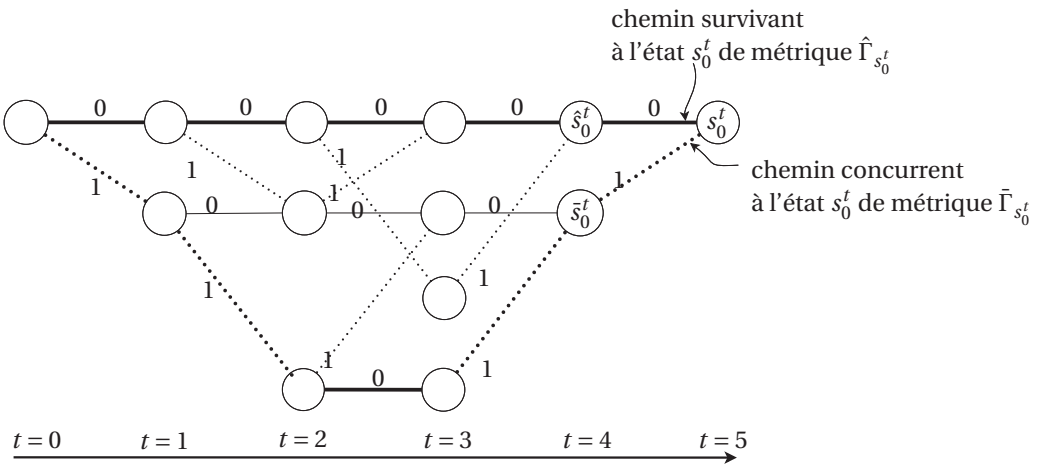

Figure 4.1. Illustration des notations

Pour déterminer la liste des i chemins les plus probables $(p_1, p_2, \ldots, p_i)$ dans le treillis T, l'algorithme SLVA effectue une phase en-avant ou forward sur T similaire à celle effectuée par l'algorithme de Viterbi(voir chapitre 1) suivie par une succession de i phases en-arrière ou backward.

Pendant la phase forward l'algorithme stocke pour chaque état s_j^t, les deux métriques $\hat{\Gamma}_{s_j^t}$ et $\bar{\Gamma}_{s_j^t}$ et les deux états $\hat{s}_j^t$ et $\bar{s}_j^t$. A la fin de cette phase forward, l'algorithme effectue une succession de i phases backward pour trouver les i chemins les plus probables $(p_1, p_2, \ldots, p_i)$.

L'implémentation de la phase forward est très simple et similaire à celle de l'algorithme de Viterbi présenté dans le paragraphe 1.4.1 dans le premier chapitre.

Pour l'implémentation des phases backward, nous allons présenter une méthode proposée par Röder *et al* dans [82]. Le choix de cette méthode est motivé par sa complexité réduite mais aussi par sa capacité à expliciter d'une manière formelle l'algorithme SLVA.

Nous allons tout d'abord montrer d'une manière simple comment les 3 premiers chemins les plus probables dans le treillis T sont obtenus par cette méthode avant de donner une description plus générale qui permet de trouver le $i^{\text{ème}}$ chemin le plus probable.

Le processus de recherche des i chemins les plus probables $(p_1, p_2, \ldots, p_i)$ commence avec la recherche du 1er chemin p_1 en effectuant une première phase backward. Cette première phase backward est similaire à celle effectuée par l'algorithme de Viterbi (voir paragraphe 1.4.1 du premier chapitre).

Pendant cette phase, la méthode proposée dans [82] définit une pile d'éléments dans laquelle sont stockés les chemins concurrents du chemin p_1: entre autres la métrique du chemin, instant où il diverge du chemin p_1. Cette pile est maintenue triée par ordre croissant selon les métriques des chemins. A la fin de la première phase backward, la méthode détermine le chemin p_1 mais aussi ses chemins concurrents stockés dans la pile.

Le 2ème chemin le plus probable p_2 est le chemin concurrent de p_1, qui a la plus grande métrique c'est-à-dire le chemin en tête de la pile. Soit t_1 l'instant du treillis T où il diverge du chemin p_1. Entre cet instant t_1 et la fin du treillis T les deux chemins p_1 et p_2 sont superposés, c'est à dire $p_2(l) = p_1(l)$, pour $l \geq t_1$. Pour tracer l'autre partie du chemin p_2, pour $l < t_1$, une deuxième phase backward est effectuée à partir de l'instant t_1. Pendant cette deuxième phase, les chemins concurrents de p_2 et qui y dvergent avant l'instant t_1 sont stockée dans la pile tout en la gardant triée par ordre croissant selon les métriques des chemins. Le chemin p_2 doit être effacé de la pile directement après avoir été extrait mais les autres chemins stockés pendant la phase backward précédente ne sont pas effacés car ils sont des chemins concurrents de p_2 et peuvent inclure le 3ème chemin le plus probable.

Le 3ème chemin le plus probable p_3 est le chemin concurrent de p_2, qui a la plus grande métrique c'est-à-dire le chemin en tête de la pile et soit t_2 l'instant du treillis T où il diverge du chemin p_2. Entre cet instant t_2 et la fin du treillis T les deux chemins p_2 et p_3 sont superposés, c'est à dire $p_3(l) = p_2(l)$, pour $l \geq t_2$. Pour tracer l'autre partie du chemin p_3, pour $l < t_2$, une troisième phase backward est effectuée à partir de l'instant t_2. Pendant cette phase backward, les chemins concurrents de p_3 et qui y divergent avant l'instant t_2 sont stockés dans la pile tout en la gardant triée par ordre croissant.

Le processus est répété pour trouver les autres chemins les plus probables: p_4, $p_5, \ldots, p_i$.

Pour décrire formellement les calculs effectués par la méthode pour trouver le $i^{\text{ème}}$ chemin le plus probable dans le treillis T, nous définissons une pile d'éléments $S_h = (S_h.m, S_h.q, S_h.t, S_h.u)$ telle que:

- $S_h.m$: est la métrique du chemin concurrent à stocker dans la pile.
- $S_h.i$: est le numéro du backward ($i^{\text{ème}}$ chemin) à lequel l'élément S_h est inséré dans la pile.
- $S_h.t$: est l'instant du treillis où le chemin à stocker diverge du chemin précédent. C'est l'instant où le backward suivant va commencer si la métrique de ce chemin à stocker est la plus grande dans la pile.
- $S_h.u$: est la métrique du chemin partiel entre l'état où le chemin concurrent à stocker diverge du chemin précédent et l'état final du treillis T. Cette métrique est nécessaire car elle permet de calculer la métrique globale du chemin concurrent à stocker et pour lequel nous ne disposons que de sa métrique partielle jusqu'à l'état où il diverge du chemin précédent.

Chaque élément S_h de la pile définit un chemin concurrent du $(i-1)^{\text{ème}}$ chemin le plus probable.

La description formelle de l'implémentation de l'algorithme SLVA pour déterminer une liste des i chemins les plus probables dans le treillis T est donnée ci-dessous [82]:

1. Dans la première phase backward, la pile est déclarée et initialisée. Cette phase trace le $1^{\text{ème}}$ chemin le plus probable ($i = 1$) dans le treillis T, en commençant de son état final $s_n^0 = 0$ du dernier étage $t = n$. Cet état est inséré dans le vecteur des états du chemin recherché p_1, $p_1(t) = 0$. Puis définir et initialiser la métrique du chemin partiel $m = 0$, l'indice d'état $l = 0$ et ensuite aller à l'étape 6. Dans les backwards suivants, mettre $i = i + 1$ et aller à l'étape 2.

2. Le chemin p_{i-1} précédent se trouve jusqu'ici stocké en tête de la pile à l'emplacement S_1. Le nouveau chemin recherché p_i diverge du chemin p_{i-1} à l'état $j = p_{i-1}(t)$ de l'étage $t = S_1.t$, et son vecteur d'états, entre les instants t et n, est rempli par les états de p_{i-1} tels que: $p_i(v) = p_{i-1}(v)$, pour $v \geq t$.
3. Mettre $m = S_1.u$. Puis supprimer le premier élément S_1 de la pile.
4. Mettre $j = \bar{s}_l^t$ ($\bar{s}_l^t$ représente la valeur décimale de l'étiquette de l'état).
5. Mettre $m = m + \bar{\Gamma}_{s_l^t} - \hat{\Gamma}_{s_j^{t-1}}$ et $p_i(t-1) = j$. Ensuite, mettre $t = t-1$ et $l = j$.
6. Insérer dans la pile un nouvel élément S_h tel que: $S_h.m = \bar{\Gamma}_{s_l^t} + m$, $S_h.i = i$, $S_h.t = t$ et $S_h.u = m$ de telle sorte à ce qu'elle reste triée en ordre croissant suivant les métriques $S_h.m$ des chemins.
7. Mettre $j = \hat{s}_l^t$ ($\hat{s}_l^t$ représente la valeur décimale de l'étiquette de l'état).
8. Mettre $m = m + \hat{\Gamma}_{s_l^t} - \hat{\Gamma}_{s_j^{t-1}}$ et $p_i(t-1) = j$. Mettre $t = t-1$ et $l = j$.
9. Répéter les étapes 6 à 8 jusqu'à $t = 0$.

La pile est triée en ordre croissant avec les métriques des chemins. Le meilleur chemin suivant se trouve donc en tête de la pile après chaque backward. Pour réduire l'espace mémoire utilisé et éviter des insertions inutiles dans la pile, la taille de celle-ci peut être limitée à $(i-1)$ éléments car tous les éléments en dessous de l'élément $(i-1)$ ne peuvent pas monter en tête de la pile et donc ne sont pas parmis les i chemins les plus probables recherchés.

4.2.2 Description de l'algorithme de décodage itératif dérivé de l'algorithme List-Viterbi

La description des différentes étapes de l'algorithme de décodage itératif dérivé l'algorithme List-Viterbi est donnée par:

- **Etape** 1**:** effectuer un décodage de Viterbi sur chacun des n_p treillis-produits T_{p_i}, $i = 0, 1, \ldots, n_p - 1$. Constituer une liste L_s de n_p mots candidats correspondant aux

chemins survivants sur les treillis-produits (un mot par treillis-produit). Mettre $i = 1$.

- **Etape** 2**:** tester s'il y a un mot valide du code $\mathscr{C}$ dans la liste L_s? si oui, retenir ce mot comme décision et stopper l'algorithme. Sinon, mettre $i = i + 1$ et aller à l'étape 3. A ce niveau une itération est effectuée.
- **Etape** 3**:** chercher sur chaque treillis-produit le $i^{\text{ème}}$ chemin le plus probable en utilisant l'algorithme SLVA précédent. Une liste L_s de n_p mots candidats est alors constituée. Aller à l'étape 2.

L'étape 2 peut être implémentée simplement par le calcul de syndrome $s = cH^T$, où H^T est la matrice transposée de H. L'étape 3 est implémentée à l'aide de l'algorithme SLVA présenté dans le paragraphe précédent. Le processus de décodage continue jusqu'à ce qu'un mot valide soit trouvé ou un nombre maximal d'itérations est atteint.

4.2.3 Performances de l'algorithme

Dans ce paragraphe, nous présentons les résultats de simulations obtenus pour les 3 codes de Hamming$(8,4,4)$, BCH$(31,26,3)$ et BCH$(32,26,4)$ dans le cas d'un canal perturbé par un bruit blanc additif gaussien ou AWGN. Les treillis-produits ont été construits par groupement de 2 lignes de la matrice H ($n_L = 2$). Cela conduit à des treillis-produits à 4 états. Pour le code de Hamming$(8,4,4)$, la structure de décodage est composée de 2 treillis-produits de même longueur $n = 8$ sections. Pour le code $(31,26)$, la structure de décodage est composée de 3 treillis-produits de même longueur $n = 31$ sections. Pour ce code le nombre de lignes de sa matrice de contrôle est impaire ($(n-k) = 5$) et donc ne divise pas $n_L = 2$. Dans ce cas, une ligne de la matrice est dupliquée pour que le nombre total des lignes soit paire.

La Figure 4.2 présente les performances de l'algorithme pour le code de Hamming$(8,4,4)$ comparées à celles du décodage ML-exhaustif. Les performances sont évaluées en termes de taux d'erreur binaire ou BER (Bit Error Rate) en fonction du rapport signal à bruit.

Nous constatons que l'algorithme réalise des performances quasi-optimales au sens ML-exhaustif avec seulement 3 itérations. Les performances sont aussi quasi-optimales avec 2 itérations.

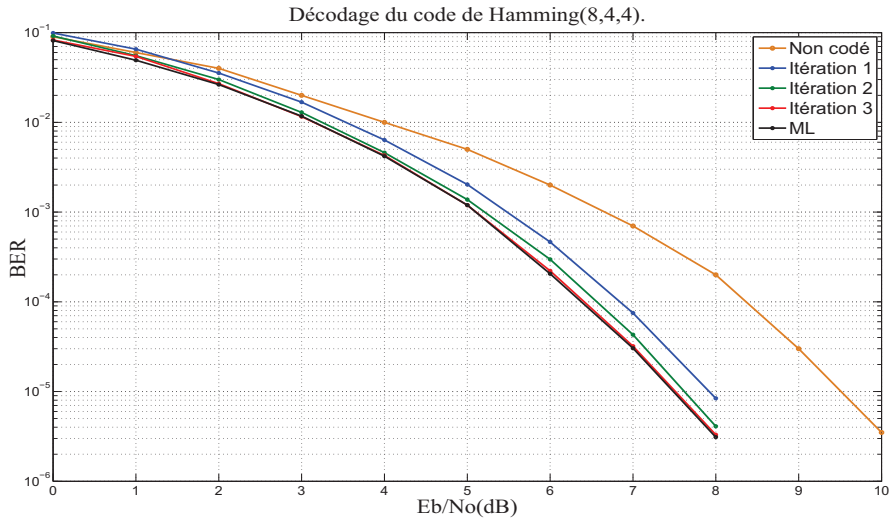

Figure 4.2. Performances de l'algorithme de dcodage itratif driv de List-Viterbi pour le code de Hamming$(8,4,4)$.

La Figure 4.3 présente les performances, évaluées en termes de BER, de l'algorithme pour le code de BCH$(31,26,3)$. Au bout de 5 itérations, ces performances sont quasi-optimales comparées à celle du décodage ML-exhaustif.

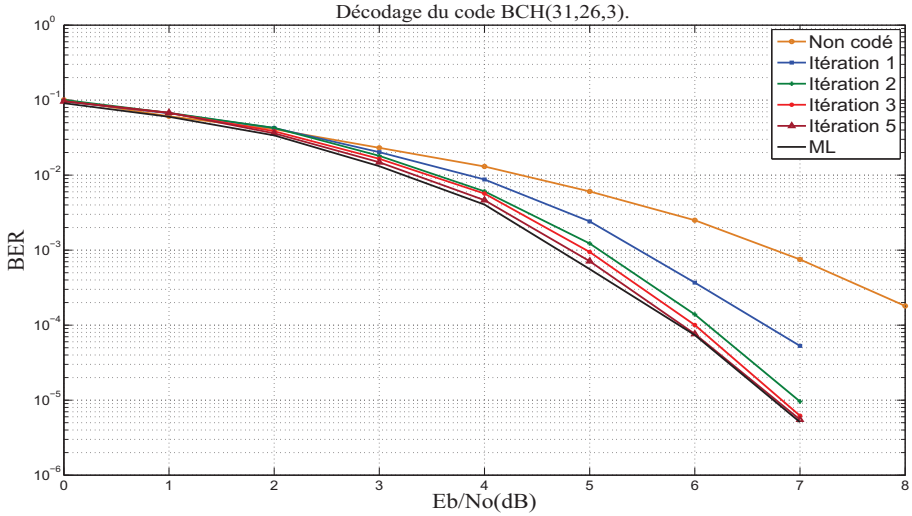

Figure 4.3. Performances de l'algorithme de dcodageitratif driv de List-Viterbi pour le code $(31,26,3)$.

La Figure 4.4 présente les performances de l'algorithme pour le code de BCH$(32,26,4)$. Au bout de 7 itérations, ces performances sont très proches de celle du décodage ML-exhaustif.

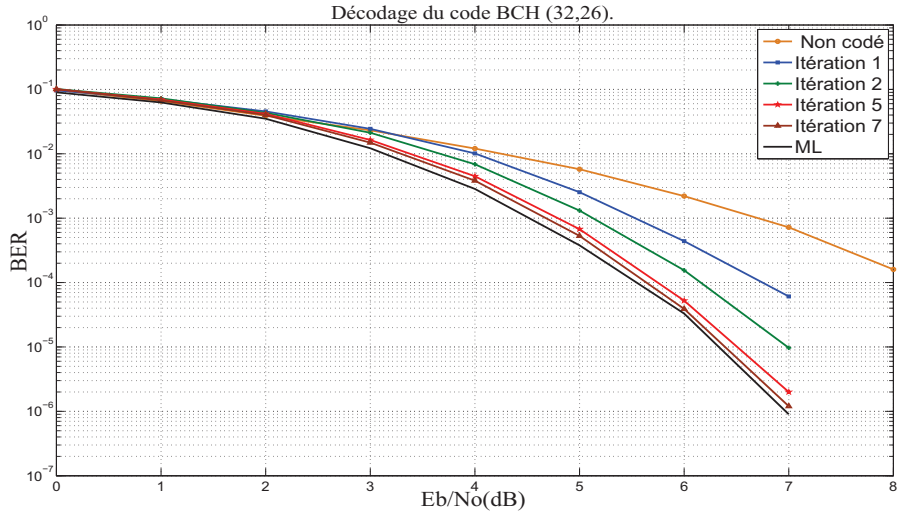

Figure 4.4. Performances de l'algorithme de dcodage itratif driv de List-Viterbi pour le code $(32,26,4)$.

Nous constatons en observant les performances de l'algorithme pour les deux codes BCH(31,26,3) et BCH(32,26,4) que sa vitesse de convergence décroit avec l'augmentation de la longueur du mot de code. En effet, pour le code BCH(31,26,3), les performances de l'algorithme sont très proches de celles du décodage ML-exhaustif en effectuant seulement 5 itérations alors que pour le code BCH(32,26,4) il fallait 7 itérations pour approcher les performances optimales.

Le principe de l'algorithme consiste à faire une recherche du chemin le plus probable sur plusieurs treillis-produits. Cela signifie que la recherche est d'autant difficile que le nombre de chemins dans ce treillis est grand. Or, le nombre de chemins (est égal à 2^{30}) dans un treillis-produit issu de la matrice de contrôle du code BCH(32,26,4) est 2 fois plus grand que le nombre de chemins(est égal à 2^{29}) dans un treillis-produit issu de la matrice de contrôle du code BCH(31,26,3). Cela justifie la convergence moins rapide de l'algorithme pour le décodage du code BCH(32,26,4). La vitesse de convergence de l'algorithme peut être accélérée en augmentant la taille des treillis-produits c'est-à-dire les formant avec plus de 2 lignes de la matrice de contrôle car cela diminue le nombre de chemins dans le treillis-produit et rend plus rapide la recherche des chemins les plus probables. Par exemple, pour le décodage du code BCH(32,26,4) les treillis-produits construits avec 3 lignes de sa matrice de contrôle contiennent 2 fois moins de chemins que les treillis-produits construits avec 2 lignes et par conséquent conduisent à une vitesse de convergence plus grande.

La convergence de l'algorithme dépend aussi du rapport signal à bruit ou SNR du canal. En effet, pour des valeurs de SNR grandes, l'algorithme trouve rapidement le chemin le plus probable sur les treillis-produits. Mais pour des valeurs de SNR petites, l'algorithme peut effectuer un grand nombre d'itérations avant de trouver le chemin le plus probable. Ces réalités sont déduites des résultats de simulations présentés sur les figures 4.5 et 4.6 qui donnent le nombre moyen d'itérations nécessaires pour trouver un mot de code valide pour les 2 code de Hamming(8,4,4) et BCH(31,26,3) respectivement.

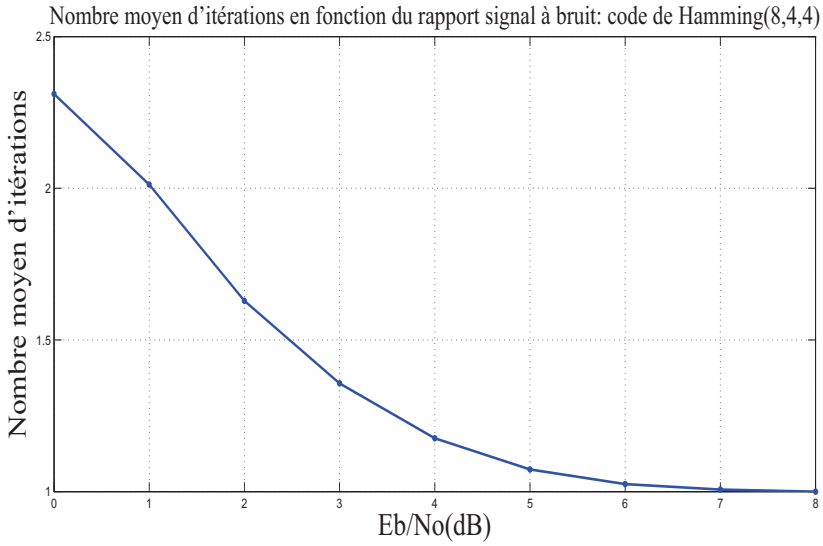

Figure 4.5. Nombre moyen d'itrations en fonction du SNR pour le code de Hamming$(8,4,4)$**.**

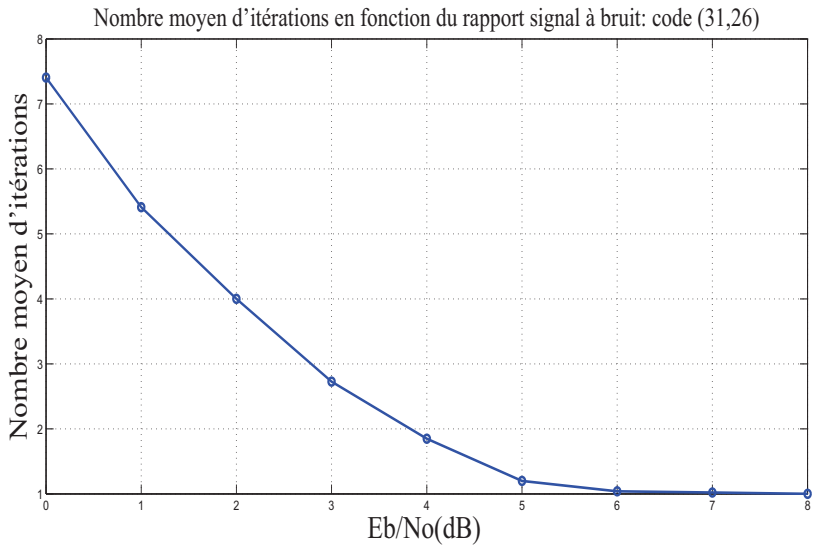

Figure 4.6. Nombre moyen d'itrations en fonction du SNR pour le code $(31,26,3)$**.**

4.2.4 Complexité de l'algorithme

La complexité de cet algorithme de décodage est la complexité d'un décodeur élémentaire (treillis-produit) multipliée par le nombre total de ces décodeurs n_p. A l'itération i, un décodeur élémentaire cherche le i$^{\text{ème}}$ chemin le plus probable sur le treillis-produit correspondant. Dans ce cas, l'algorithme effectue l'étape 1 seule, suivie par $(i-1)$ itérations des étapes 2 et 3. Le décodage termine avec l'étape 2.
L'étape 1 consiste à effectuer un décodage de Viterbi sur chacun des treillis-produits. Dans cette étape, l'algorithme de Viterbi est appliqué sur n_p treillis-produits à 4 états, donc la complexité de la première étape est de l'ordre de $n_p \times n \times 2^4$. Le test des mots à l'étape 2 est effectué par le calcul de syndrome $s = cH^T$. Cela peut être fait par le calcul des syndromes élémentaires, impliquant seulement des opérations binaires. La complexité de cette étape est négligeable devant la complexité de l'étape 1 et 3, qui implique des additions et comparaisons. L'implémentation de l'algorithme SLVA, dans l'étape 3, nécessite une pile qui grandit linéairement avec chaque itération. Pour une phase backward, l'insertion d'un élément dans la pile nécessite, en moyenne, $i/2$ comparaisons. Si on suppose que le $i^{\text{ème}}$ chemin recherché diverge du chemin précédent, en moyenne, au milieu du treillis ($t = n/2$), alors il y a $n/2$ états à visiter dans chaque backward. Ceci conduit à $n/2$ insertions dans la pile et $3n/2$ additions. Pour i chemins, la complexité globale de l'étape 3 est de l'ordre de $\mathcal{O}(n(i^2+i)) = \mathcal{O}(ni^2)$. Donc, la complexité globale de la méthode de décodage est de l'ordre $\mathcal{O}(n_{p_H} ni^2))$.

4.3 Décodage itératif SISO basé sur des treillis-produits

La technique de décodage itératif présentée dans ce paragraphe est un algorithme SISO qui estime les LLRs *a posteriori* des symboles reçus. Cet algorithme, comme le précédent, utilise des treillis-produits de complexité réduite construits à partir des treillis élémentaires représentant les lignes de la matrice de contrôle du code. L'algorithme fait coopérer ces treillis-produits dans un processus itératif pendant lequel ils échangent

des informations extrinsèques sur les symboles formant les étiquettes des branches de leurs sections. Ces informations extrinsèques sont calculées sur chaque treillis-produit en utilisant un algorithme SISO comme l'algorithme BCJR[34].

L'idée de départ de cet algorithme de décodage est d'utiliser la matrice de contrôle pour former les treillis-produits utilisés dans la structure de décodage. L'application de cet algorithme pour le décodage de certains codes en bloc a ensuite donné de bonnes performances nous encourageant à aller davantage dans cette direction. L'application de l'algorithme à d'autres codes en bloc a rapidement révélé des inconvénients liés à la présence des sections nulles sur les treillis-produits qui empêchent certains symboles de profiter du décodage itératif. Ce problème rend très difficile d'approcher les performances optimales avec une complexité raisonnable pour autres codes comme le code de Golay(24,12,8).

Nous avons ensuite orienté nos recherches sur ce problème posé par les sections nulles afin d'en trouver des solutions. Nous avons donc abouti à deux solutions. La première solution consiste à utiliser une autre matrice de contrôle du code conjointement avec la matrice de contrôle de départ afin d'assurer une couverture des symboles présents dans ces sections nulles. La deuxième solution consiste à faire une sectionnalisation des treillis-produits en fusionnant les sections nulles avec d'autres sections non-nulles pour former une seule section.

Cette section est organisée comme suit. La sous-section 4.3.1 donne une description générale de l'algorithme en présentant sa structure graphique avec une description formelle de toutes ses étapes. La sous-section 4.3.2 décrit la notion de sectionnalisation des treillis et l'expose à l'aide d'un exemple d'un treillis simple. La sous-section 4.3.3 présente une modification de l'algorithme BCJR adaptée aux treillis sectionnalisés. La sous-section 4.3.4 donne un exposé d'exemple de construction de la structure de décodage à l'aide du code de Hamming $(7,4,3)$ et présente les performances de l'algorithme pour plusieurs codes en bloc. La sous-section 4.3.5 calcule la complexité de l'algorithme. Cet algorithme a fait l'objet de la publication [83].

4.3.1 Notations et description générale de l'algorithme

La structure de décodage de l'algorithme est constituée d'un ensemble de treillis-produits. Si ces treillis-produits sont sectionnalisés, leurs branches sont étiquetées par un ensemble de bits. Chaque branche d'une section t d'un treillis-produit est étiquetée par un groupe de m_t bits qui forme un symbole. Dans ce cas les treillis-produits échangent des informations extrinsèques sur les symboles et non sur les bits. Par contre si les treillis-produits ne sont pas sectionnalisés, ils échangent des informations sur les bits étiquetant leurs branches.

Pour pouvoir donner une description générale de l'algorithme qui soit valable pour les deux cas ci-dessus des treillis-produits, nous pouvons considérer le deuxième cas comme étant un cas particulier du premier avec $m_t = 1$ et nous utilisons, dorénavant, l'appellation symbole pour décrire soit un bit soit un groupe de bits.
Pour un symbole $h^{(t)}$, nous notons par $L_a(h^{(t)})$, son LLR *a priori* donné par:

$$L_a(h^{(t)}) = \log \frac{P(h^{(t)}|y^{(t)})}{P(0|y^{(t)})} \tag{4.2}$$

où $y^{(t)}$ est la séquence partielle du mot reçu y associée aux m_t bits de code. Ce vecteur des LLR *a priori* sera délivré à chaque treillis-produit pour initialiser le décodage.
Les décodeurs constituant la structure globale de décodage sont représentés par les treillis-produits T_{p_d}, $d = 0, 1, \ldots, n_p - 1$, où n_p est le nombre total des treillis-produits. On note par L_e^d, le vecteur des LLR extrinsèques fourni, après chaque itération, par le treillis-produit T_{p_d} sur les symboles et par L_a^d, le vecteur des LLR des symboles fourni au treillis-produit T_{p_d} pour commencer une nouvelle itération de décodage.

La structure générale de décodage pour le code en bloc $\mathscr{C}$ est décrite par la Figure 4.7.

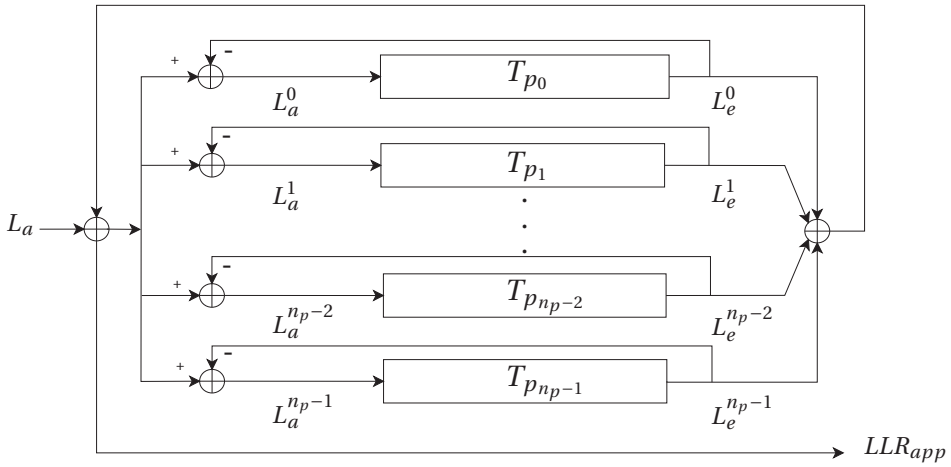

Figure 4.7. Structure graphique de dcodage

Une description formelle de cette méthode de décodage est donnée ci-après:

Algorithm: décodage itératif SISO par treillis-produits

Input : L_a(LLR a priori des symboles)

$L_a^d \longleftarrow L_a, d \in \{0,1,\ldots,n_p-1\}$;

for $nIter \leftarrow 1$ **to** $nombreMaxIterations$ **do**

  for $d \leftarrow 0$ **to** n_p-1 **do**

    $L_e^d \longleftarrow$ algorithme BCJR(L_a^d).

  end

  for $d \leftarrow 0$ **to** n_p-1 **do**

$$L_a^d \longleftarrow \sum_{i=0, i\neq d}^{n_p-1} L_e^i(c) + L_a.$$

  end

end

$$LLR_{app}(h \mid \mathbf{y}) \longleftarrow \sum_{d=0}^{n_p-1} L_e^d(h) + L_a(h).$$

Output: LLR_{app}(LLR a posteriori des symboles).

4.3.2 Sectionnalisation des treillis-produits

La sectionnalisation[84] d'un treillis consiste à fusionner des sections successives pour en former une seule. Chaque section résultant d'une fusion a donc pour étiquettes de branches la concaténation des étiquettes des branches de transition entre les états de départ et les états d'arrivée des sections fusionnées.

Soit T un treillis de longueur n et T_s le treillis résultant de la sectionnalisation du treillis T aux instants $(t_0, t_1, \ldots, t_v)$ où v est le nombre de sections dans le treillis T_s tels que $0 = t_0 < t_1 < \ldots < t_v = n$. Une section i dans T_s est résultante de la fusion de $m_i = t_i - t_{i-1}$ sections de T et chacune de ses branches est étiquettée par m_i bits. Pour exprimer la sectionnalisation sous forme mathématique, considérons deux sections successives S_0 et S_1 d'un même treillis T telles que: $S_0 = \{(a_0, e_0, b_0)\}$ et $S_1 = \{(a_1, e_1, b_1)\}$ où a_i, b_i et e_i, $i \in \{0,1\}$, sont respectivement les ensembles d'états de départ, d'états d'arrivée et d'étiquettes des branches constituant une section S_i. La sectionnalisation $(S_0|S_1)$ ou fusion des sections S_0 et S_1 est définie telle que:

$$(S_0|S_1) = \{(a_0, b_0), (e_0, e_1), (a_1, b_1)|b_0 = a_1\}$$

ce qui se simplifie par la suppression des produits cartésiens ci-dessus pour lesquels $b_0 \neq a_1$: $(S_0|S_1) = \{a_0, (e_0, e_1), b_1\}$.

Par exemple, le treillis T_s de 3 sections décrit par la Figure 4.9 est obtenu par la sectionnalisation du treillis T décrit par la Figure 4.8. La première section de T_s est obtenue par fusion des 2 sections 1 et 2 de T, la deuxième section par fusion des 2 sections 3 et 4 de T et la troisième et dernière section de T_s est obtenue par fusion des 3 sections 5, 6 et 7 de T. Si le nombre de section groupées est constant, nous parlons d'une sectionnalisation invariante.

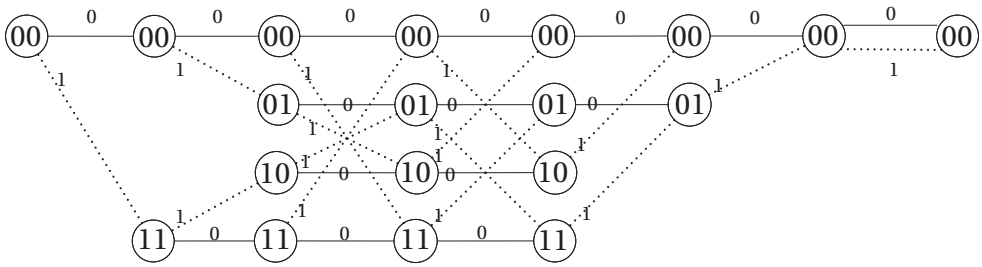

Figure 4.8. Treillis T.

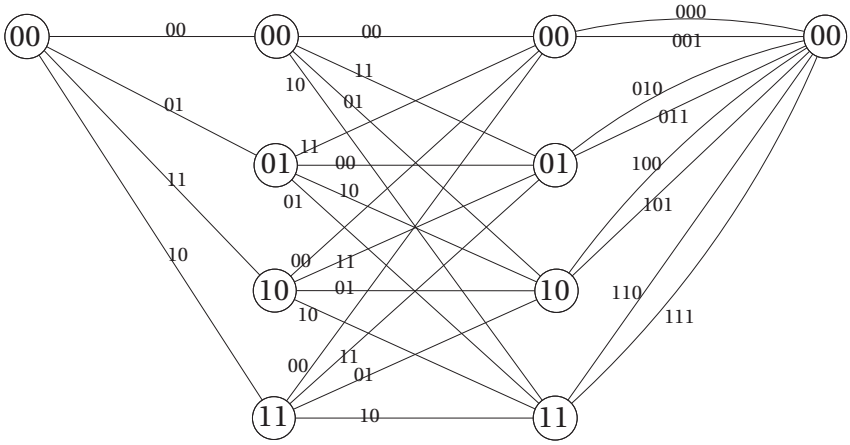

Figure 4.9. Treillis T_s obtenu par sectionnalisation du treillis T dcrit par la Figure 4.8.

La sectionnalisation d'un treillis-produit peut conduire à la formation de branches multiples entre deux états du treillis comme dans la dernière section du treillis-produit présenté sur la Figure 4.9 où deux états sont connectés par 2 branches distinctes. La formation des branches multiples peut être exploitée pour réduire la complexité de décodage de l'algorithme BCJR [85] et donc la complexité globale de l'algorithme proposé. Les branches multiples peuvent être fusionnées en une simple branche. Ceci réduit le nombre total de branches dans le treillis et donc réduit la complexité globale

du décodage (voir paragraphe 1.2.1.5). La Figure 4.10 illustre le passage d'un treillis à branches multiples à un treillis à branches simples.

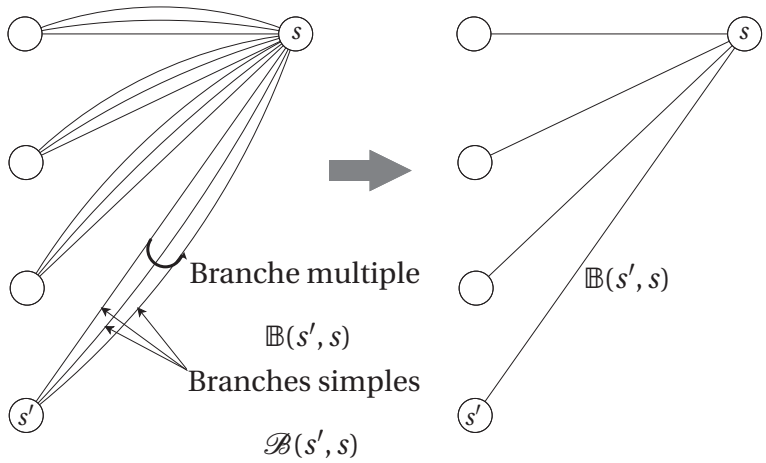

Figure 4.10. Transformation de branche multiple en une seule branche.

Il est donc très commode pour notre application d'évaluer la complexité de branche d'un treillis sectionnalisé par le nombre de branches multiples. Pour cela nous introduisons tout d'abord quelques notations. On note par $\mathscr{C}_T$ le code en bloc décrit par le treillis T et $\mathscr{C}_{t_{i-1},t_i}$ le sous-code du $\mathscr{C}_T$ contenant les mots de code dont les bits non nuls se trouvent entre les positions $(t_{i-1}+1)$ et t_i, $\mathscr{C}_{p_{t_{i-1},t_i}}$, le code poinçonné de longueur m_i obtenu du code $\mathscr{C}_T$ en supprimant les premiers t_{i-1} et derniers t_i bits de chaque mot de code du $\mathscr{C}_T$. Pour un code en bloc $\mathscr{C}$, on note par $k(\mathscr{C})$ sa dimension. Le nombre total d'états n_s^i à l'étage t_i du treillis T_s est donné par[84]:

$$n_s^i = 2^{k(\mathscr{C}_T)-k(\mathscr{C}_{0,t_i})-k(\mathscr{C}_{t_i,n})} \tag{4.3}$$

Le nombre de branches multiples arrivant en un état s_i, $deg(s_i)_{in}$, et le nombre des branches quittant cet état, $deg(s_i)_{out}$, sont donnés par[84]:

$$deg(s_i)_{in} = 2^{k(\mathscr{C}_{0,t_i})-k(\mathscr{C}_{t_i,n})-k(\mathscr{C}_{t_{i-1},t_i})} \tag{4.4}$$

$$deg(s_i)_{out} = 2^{k(\mathscr{C}_{t_{i-1},n})-k(\mathscr{C}_{t_i,n})-k(\mathscr{C}_{t_{i-1},t_i})} \tag{4.5}$$

La complexité de branche d'une section i du treillis T_s est donnée par les paramètres suivants[84]:

1. La taille de la branche multiple c'est-à-dire le nombre de branches simples constituant la branche multiple:
$$N_b^i = 2^{k(\mathscr{C}_{t_{i-1},t_i})} \tag{4.6}$$

2. Le nombre de branches multiples distinctes:
$$N_d^i = 2^{k(\mathscr{C}_{p_{t_{i-1},t_i}})-k(\mathscr{C}_{t_{i-1},t_i})} \tag{4.7}$$

3. Et finalement le nombre total des branches multiples:
$$\begin{aligned} N_B^i &= n_{s_i}.deg(s_i)_{in} \\ &= 2^{k(\mathscr{C}_T)-k(C_{t_i,n})-k(\mathscr{C}_{0,t_{i-1}})-k(\mathscr{C}_{t_{i-1},t_i})} \end{aligned} \tag{4.8}$$

D'après les équations 4.7 et 4.8, le nombre d'occurrences de chaque branche multiple dans la section i, q_d^i, est donné par:
$$q_d^i = 2^{k(\mathscr{C}_T)-k(\mathscr{C}_{0,t_{i-1}})-k(\mathscr{C}_{t_i,n})-k(\mathscr{C}_{p_{t_{i-1},t_i}})} \tag{4.9}$$

En général le nombre total de branches multiples N_B^i est beaucoup plus grand que le nombre des branches multiples distinctes N_d^i. Cette réalité peut être exploitée pour réduire la complexité de décodage de l'algorithme en calculant, dans une phase de prétraitement, les probabilités des branches distinctes et les stocker ensuite dans une table de correspondance.

La sectionnalisation d'un treillis-produit peut se faire de plusieurs manières. La sectionnalisation qui conduit à un nombre minimal d'opérations arithmétiques et d'allocations mémoire est appelée sectionnalisation optimale. Deux algorithmes pour trouver la sectionnalisation optimale d'un treillis d'un code en bloc sont proposés dans [85][86]. Plusieurs exemples de sectionnalisations optimales pour certains codes en bloc sont proposées dans [37, 58][85, 86].

4.3.3 *Algorithme BCJR sur un treillis-produit sectionnalisé*

Dans le treillis-produit sectionnalisé, chaque branche est étiquetée d'un groupe de bits. Ce groupe de bits est traité par l'algorithme BCJR comme étant un symbole indissociable. Dans ce cas, les treillis-produits échangent des informations extrinsèques sur les symboles et pas sur les bits. Soit m_t le nombre de bits par symbole, et $y^{(t)} = (y_{tm_t}, y_{tm_t+1}, \ldots, y_{m_t(t+1)-1})$ la séquence partielle de y correspondant aux m_t bits présents à l'instant t du treillis-produit sectionnalisé. La probabilité d'une branche, $\mathscr{B}(s', s)$, connectant un état s' de l'étage $t-1$ à un état s de l'étage t et étiquetée par le symbole $h^{(t)} = (h_0^{(t)}, h_1^{(t)}, \ldots, h_{m_t-1}^{(t)})$ est donnée par:

$$\begin{aligned}\gamma_t(s', s) &= P(y^{(t)}, s|s') \\ &= P(y^{(t)}|h^{(t)})P(h^{(t)}) \\ &= P(h^{(t)})\, w_t(s', s)\end{aligned} \tag{4.10}$$

où $P(h^{(t)})$ est la probabilité *a priori* du symbole et $w_t(s', s)$ est donnée par:

$$w_t(s', s) = \prod_{m=0}^{m_t-1} P(y_{tm_t+m}|h_m^{(t)}) \tag{4.11}$$

A la première itération du décodage itératif la valeur de $P(h^{(t)})$ est égale à $\frac{1}{2^{m_t}}$ car les bits sont supposés équiprobables. A l'itération I, elle est déduite de l'information extrinsèque fournie par les autres décodeurs à l'itération $I-1$. Dans le cas d'un canal gaussien, la probabilité $w_t(s', s)$ est donnée par:

$$\begin{aligned}w_t(s', s) &= \prod_{m=0}^{m_t-1} \frac{1}{\sqrt{\pi N_0}} e^{-\frac{1}{N_0}(y_{tm_t+m} - h_m^{(t)})^2} \\ &= C_t e^{\frac{2}{N_0}\sum_{m=0}^{m_t-1} y_{tm_t+m}.h_m^{(t)}}\end{aligned} \tag{4.12}$$

où $C_t = (\frac{1}{\sqrt{\pi N_0}})^{m_t/2} e^{\sum_{m=0}^{m_t-1} -\frac{1}{N_0}(y_{tm_t+m}^2 + (h_m^{(t)})^2)}$ est une constante dans la section t. Comme la valeur de $w_t(s', s)$ sera utilisée dans un rapport de vraisemblance (équation 4.17), la constante C_t peut être ignorée et l'expression de $w_t(s', s)$ devient:

$$w_t(s', s) = e^{\frac{2}{N_0}\sum_{m=0}^{m_t-1} y_{tm_t+m}.h_m^{(t)}} \tag{4.13}$$

Le déroulement de l'algorithme BCJR sur le treillis-produit sectionnalisé commence par le calcul des probabilités des branches multiples distinctes. La probabilité d'une branche multiple est la somme des probabilités des branches simples qui la constituent. Si on désigne par $\mathbb{B}(s',s)$ la branche multiple connectant l'état s' de l'étage $t-1$ à l'état s de l'étage t et par $\mathscr{B}(s',s)$, une branche simple de probabilité $\gamma_t(s',s)$. La probabilité de la branche multiple $\mathbb{B}(s',s)$, $\Gamma_t(s',s)$, est donnée par:

$$\Gamma_t(s',s) = \sum_{\mathscr{B}(s',s)\in\mathbb{B}(s',s)} \gamma_t(s',s) \tag{4.14}$$

Dans le cas où le treillis-produit sectionnalisé ne comporte pas de branches multiples entre les deux états s' et s, la valeur de $\Gamma_t(s',s)$ est égale $\gamma_t(s',s)$. Les probabilités des branches multiples sont ensuite utilisées pour calculer les probabilités forward α_t et backward β_t comme suit:

$$\alpha_t(s) = \sum_{s'} \Gamma_t(s',s).\alpha_{t-1}(s'), \text{ avec } \alpha_0(s) = \begin{cases} 1 & \text{si } s=0 \\ 0 & \text{sinon} \end{cases} \tag{4.15}$$

$$\beta_t(s) = \sum_{s'} \Gamma_t(s',s).\beta_{t+1}(s'), \text{ avec } \beta_n(s) = \begin{cases} 1 & \text{si } s=0 \\ 0 & \text{sinon} \end{cases} \tag{4.16}$$

A la fin des 2 phases forward et backward, l'algorithme calcule les probabilités extrinsèques des symboles. Nous pouvons également utiliser les LLRs extrinsèques de ces symboles pour simplifier les calculs. Le LLR extrinsèque d'un symbole $h^{(t)}$ calculée par un décodeur d est donné par:

$$L_e^d(h^{(t)}|y) = \log\left[\frac{\sum_{\mathscr{B}(s',s)\in\mathbb{B}^{h^{(t)}}} \alpha_{t-1}(s').\beta_t(s)}{\sum_{\mathscr{B}(s',s)\in\mathbb{B}^{\mathbf{0}}} \alpha_{t-1}(s').\beta_t(s)})\right]. \tag{4.17}$$

où $\mathbb{B}^{h^{(t)}}$ (respectivement $\mathbb{B}^{\mathbf{0}}$) est l'ensemble des branches multiples de la section t du treillis où chaque branche multiple contient une branche simple étiquetée par le symbole $h^{(t)}$ (respectivement par le symbole nul). Les LLRs extrinsèques des symboles seront

ensuite échangés entre les treillis-produits pour une nouvelle itération de l'algorithme. A cette nouvelle itération, le décodeur d met à jour le LLR *a priori* de chaque symbole par la somme des LLRs extrinsèques fournis par les autres treillis-produits à l'itération précédente. Pour un symbole $h^{(t)}$, le LLR a priori $L_a^d(h^{(t)})$ est donné par:

$$\begin{aligned} L_a^d(h^{(t)}) &\triangleq \log \frac{P(h^{(t)})}{P(0)} \\ &\approx \sum_{m=0, m\neq d}^{n_p-1} L_e^m(h^{(t)}|y). \end{aligned} \tag{4.18}$$

où $P(0)$ est la probabilité *a priori* du symbole nul et $L_e^m(h^{(t)}|y)$ est le LLR extrinsèque du symbole $h^{(t)}$ fourni par le treillis-produit m selon l'équation 4.17. La probabilité *a priori* du symbole $h^{(t)}$, $P(h^{(t)})$, est directement déduite de la valeur de de son LLR *a priori* $L_a^d(h^{(t)})$ par:

$$P(h^{(t)}) = P(0).e^{L_a^d(h)}. \tag{4.19}$$

La probabilité $P(0)$ peut être ignorée car elle sera présente dans le numérateur et dénominateur du rapport de vraisemblance dans l'équation 4.17. La probabilité $P(h^{(t)})$ est ensuite intégrée dans le calcul des probabilités des branches dans l'équation 4.10 pour effectuer une nouvelle itération de l'algorithme itératif. Dans la phase finale de l'algorithme itératif où le nombre maximal d'itérations est atteint, le LLR d'un symbole $h^{(t)}$ est donné par:

$$LLR(h^{(t)}|y) \approx \sum_{m=0}^{n_p-1} L_e^m(h^{(t)}|y) + L_a(h^{(t)}) \tag{4.20}$$

Dans une section du treillis sectionnalisé, le nombre total de branches multiples est, généralement, beaucoup plus grand que le nombre des branches multiples distinctes. Dans le calcul des probabilités α et β et ensuite les LLRs, il suffit donc de calculer les probabilités des branches multiples distinctes. Nous pouvons donc faire une phase de prétraitement dans laquelle sont calculées les probabilités des branches multiples distinctes, Γ_t et stockées dans une table de correspondance.

4.3.4 Exposé d'un exemple de réalisation et performances de l'algorithme

Nous explicitons la méthode de décodage sur le code de Hamming(7,4) de paramètres ($n = 7, k = 4$) dont une matrice de contrôle, H, est donnée par:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{4.21}$$

Chaque treillis-produit est construit par groupement de 2 lignes de la matrice de contrôle. Dans le cas où le nombre de lignes de la matrice de contrôle ne divise pas 2, comme le cas du code de Hamming(7,4), la méthode duplique une ligne de la matrice pour former avec la dernière ligne un treillis-produit. Les lignes de la matrice sont représentées par des treillis élémentaire à 2 états notés respectivement T_0, T_1 et T_2. Sur les figures présentant les treillis élémentaires et treillis-produits, une ligne solide représente un bit 1 tandis qu'une ligne pointillée représente un bit 0.

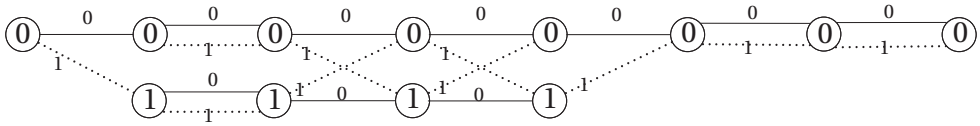

Figure 4.11. Treillis Imentaire T_0 associ au vecteur ligne 1011100.

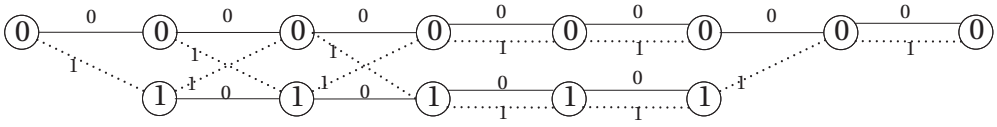

Figure 4.12. Treillis Imentaire T_1 associ au vecteur ligne 1110010.

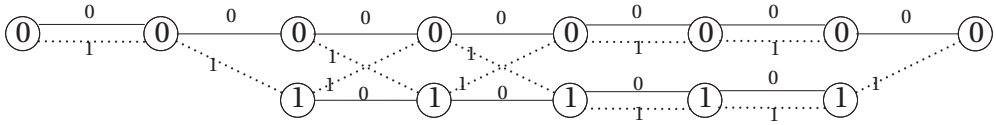

Figure 4.13. Treillis Imentaire T_2 associ au vecteur ligne 0111001.

La structure de décodage du code de Hamming(7,4) est donc composée de 2 décodeurs élémentaires représentés par les treillis-produits $T_{p_0} = T_0 \otimes T_1$ et $T_{p_1} = T_0 \otimes T_2$ présentés respectivement sur les figures 4.14 et 4.15.

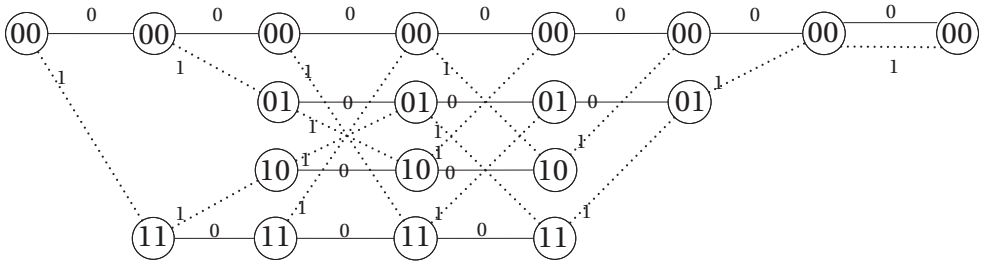

Figure 4.14. Treillis-produit $T_{p_0} = T_0 \otimes T_1$.

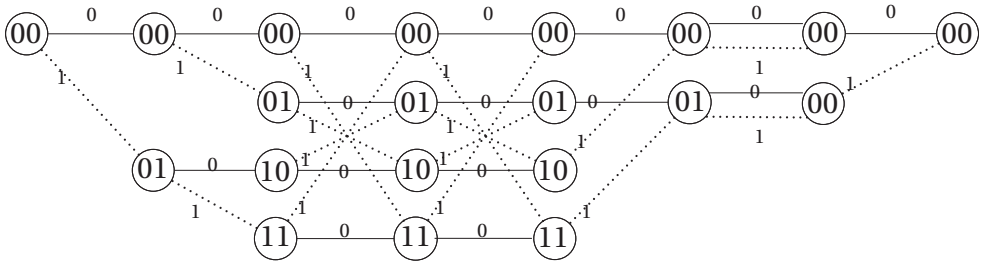

Figure 4.15. Treillis-produit $T_{p_1} = T_0 \otimes T_2$.

La structure graphique globale du code de Hamming(7,4,3) est décrite par la Figure ci-dessous.

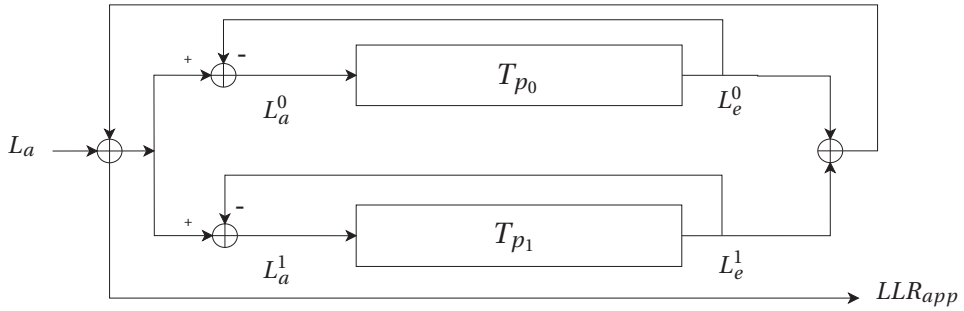

Figure 4.16. Structure graphique de dcodage pour le code de Hamming(7,4,3)

Les performances de décodage de l'algorithme pour ce code de Hamming(7,4,3) sont quasi-optimales au sens du décodage ML-exhaustif. Les figures 4.17 et 4.18 présentent respectivement ces performances comparées à celles du décodage optimal ML-exhaustif dans les deux cas de canal: un canal perturbé par un bruit blanc additif gaussien ou AWGN et un canal de Rayleigh. Nous constatons en observant les courbes de taux d'erreur binaire présentées sur ces figures que les performances de l'algorithme atteignent rapidement l'optimal au bout de 3 itérations. Même dès la première itération, les performances sont déjà très proches de l'optimal.

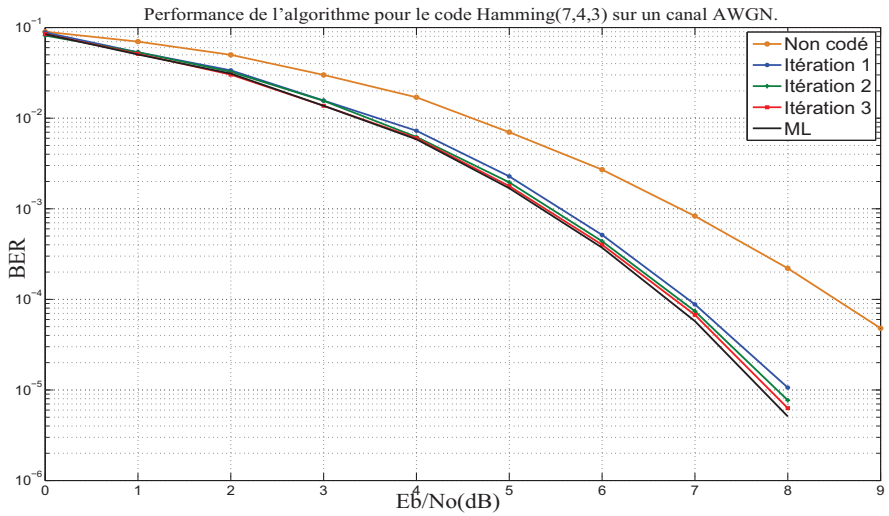

Figure 4.17. Performances de l'algorithme pour le code de Hamming$(7,4,3)$ sur un canal AWGN.

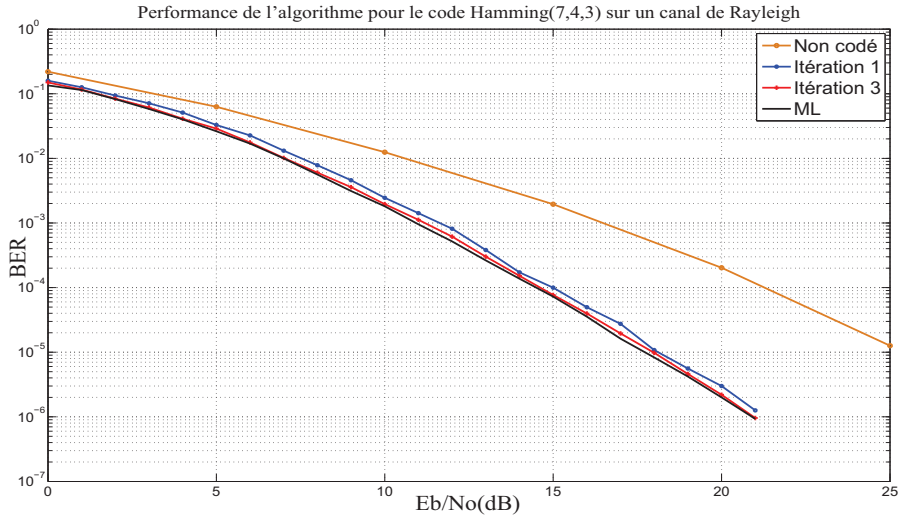

Figure 4.18. Performances de l'algorithme pour le code de Hamming$(7,4,3)$ sur un canal de Rayleigh.

Les figures 4.19 et 4.20 présentent respectivement les performances de l'algorithme, pour le code de Hamming(15,11,3), comparées à celles du décodage optimal ML-exhaustif dans les deux cas de canaux AWGN et Rayleigh.

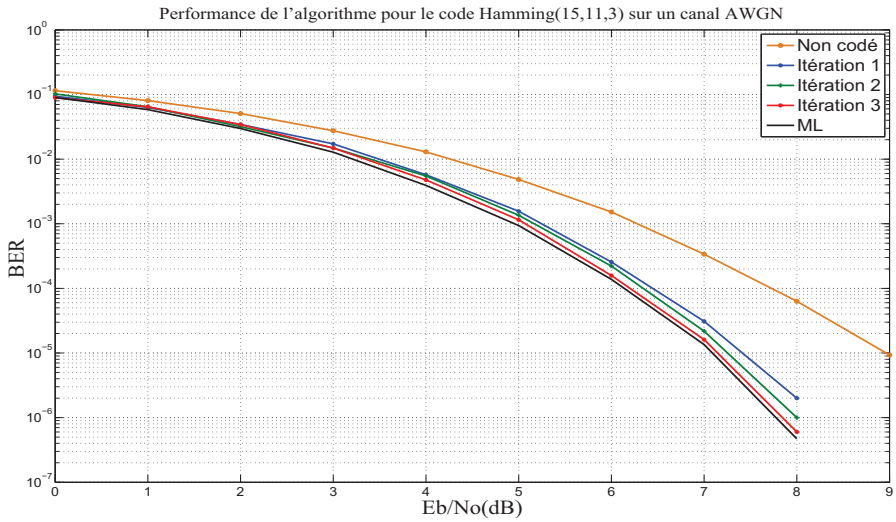

Figure 4.19. Performances de l'algorithme pour le code de Hamming$(15,11,3)$ sur un canal AWGN.

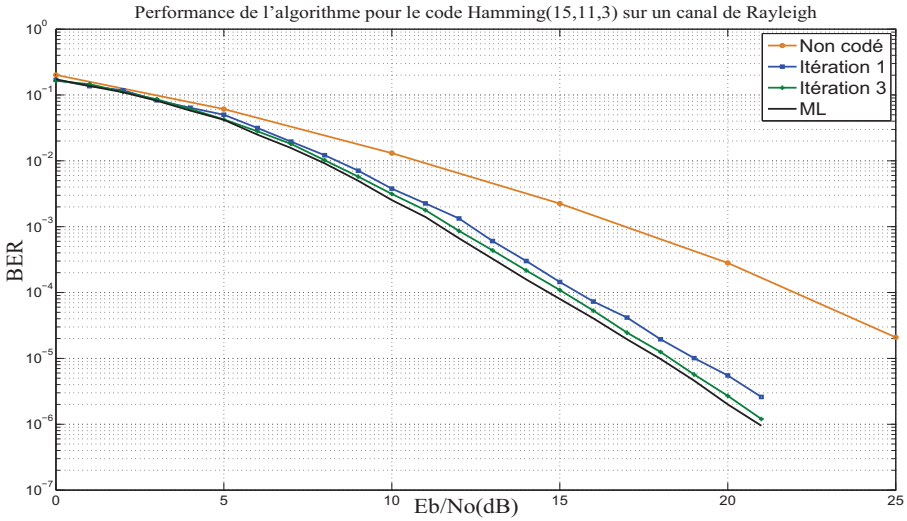

Figure 4.20. Performances de l'algorithme pour le code de Hamming $(15,11,3)$ **sur un canal de Rayleigh.**

La vitesse de convergence de l'algorithme pour le décodage des deux codes de Hamming ci-dessus est plutôt rapide car au bout de 3 itérations seulement, ses performances sont similaires à celle du décodage optimal ML-exhaustif.

L'algorithme ne garde, malheureusement, pas le même comportement pour d'autres codes en bloc pour lesquels il doit effectuer un grand nombre d'itérations avant d'approcher les performances optimales.

En réalité, ce problème de convergence est lié, en partie, à la présence des sections nulles dans les treillis-produits. Une section nulle S_t est une section où chaque états s de l'étage $t-1$ est connecté à un état homologue s de l'étage t par l'ensemble des branches possibles. Deux états de cette section ne sont pas connectés entre eux s'ils sont différents. A titre d'exemple, la section 7 du treillis-produit T_{p_0} et la section 6 du treillis-produit T_{p_1} présentés respectivement sur les Figures 4.14 et 4.15 sont des sections nulles. L'inconvénient d'une section nulle est que le treillis-produit ne fournit aucune information aux autres treillis-produits sur les bits étiquetant les branches de cette section, et donc certains bits du mot reçu profitent moins du décodage itératif et

ralentissent la convergence de l'algorithme.

Nous proposons dans la suite trois techniques permettant de limiter l'effet des sections nulles et d'accélérer la vitesse de convergence de l'algorithme. La première technique consiste à augmenter les tailles des treillis-produits en les formant avec plus de lignes de la matrice de contrôle. La seconde technique consiste à sectionnaliser les treillis-produits en fusionnant les sections nulles avec d'autres sections non-nulles. La troisième technique consiste à utiliser pour le décodage deux matrices de contrôle différentes pour assurer une couverture équilibrée entre tous les bits de code y compris ceux qui existent sur des sections nulles.

4.3.4.1 Effets de la taille des treillis-produits sur les performances

Les performances de l'algorithme de décodage dépend de la taille des treillis-produits c'est-à-dire du nombre de lignes n_L formant ces treillis. Cette relation peut être justifiée par le fait que l'augmentation de la taille des treillis-produits réduit le nombre des sections nulles dans ces treillis-produits et donc fait profiter davantage plus de bits au décodage itératif, ce qui accélère la convergence de l'algorithme. La relation entre les performances et les tailles des treillis-produits est aussi confirmée par les résultats de simulations. La Figure 4.21 présente les performances de l'algorithme pour le code de Golay(24,12,8) dans les 3 cas suivants:

- Les treillis-produits sont à 4 états, c'est-à-dire ils sont formés chacun par 2 lignes de la matrice de contrôle.
- Les treillis-produits sont à 8 états, c'est-à-dire ils sont formés chacun par 3 lignes de la matrice de contrôle.
- Les treillis-produits sont à 16 états, c'est-à-dire ils sont formés chacun par 4 lignes de la matrice de contrôle.

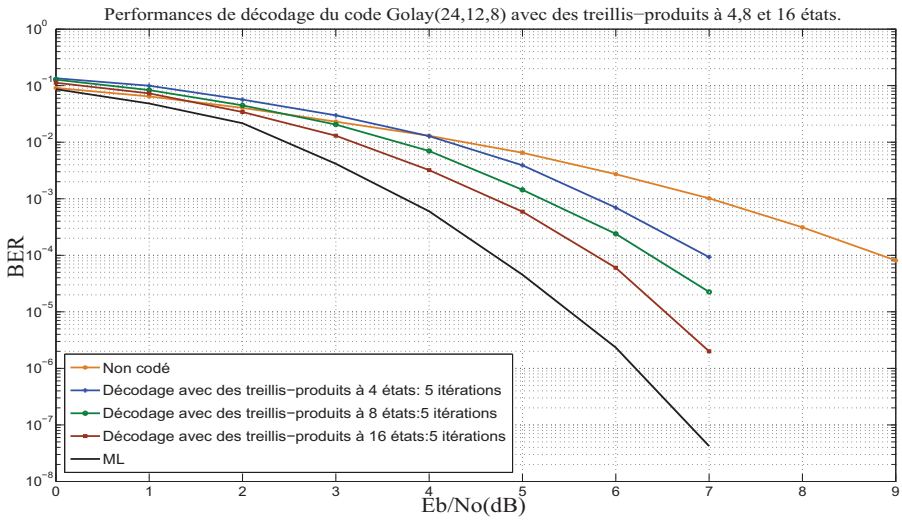

Figure 4.21. Performances de l'algorithme en fonction de la taille des treillis-produits pour le code de Golay(24,12,8).

Nous constatons en observant les courbes de taux d'erreur binaire sur la Figure 4.21 que l'augmentation des tailles des treillis-produits améliore les performances de l'algorithme. En effet, à un taux d'erreur binaire de 10^{-4}, le décodage avec des treillis-produits à 8 états apporte un gain d'environ 0.6 dB par rapport au décodage avec des treillis-produits à 4 états. Le décodage avec des treillis-produits à 16 états apporte aussi un gain d'environ 0.6 dB par rapport au décodage avec des treillis-produits à 8 états et 1.2 dB par rapport au décodage avec des treillis-produits à 4 états.

La Figure 4.22 présente les performances de l'algorithme pour le code BCH(32,26,4) dans les 2 cas suivants:

- Les treillis-produits sont à 4 états, c'est-à-dire ils sont formés chacun par 2 lignes de la matrice de contrôle.
- Les treillis-produits sont à 8 états, c'est-à-dire ils sont formés chacun par 3 lignes de la matrice de contrôle.

Encore et en observant les courbes de taux d'erreur binaire présentées sur la Figure 4.22, nous constatons que les performances de décodage avec des treillis-produits à 8 états sont nettement supérieurs à celles obtenus avec le décodage par des treillis-produits à 4 états. A un taux d'erreur binaire de 10^{-4}, le décodage avec des treillis-produits à 8 états apporte un gain d'environ 0.5 dB par rapport au décodage avec des treillis-produits à 4 états.

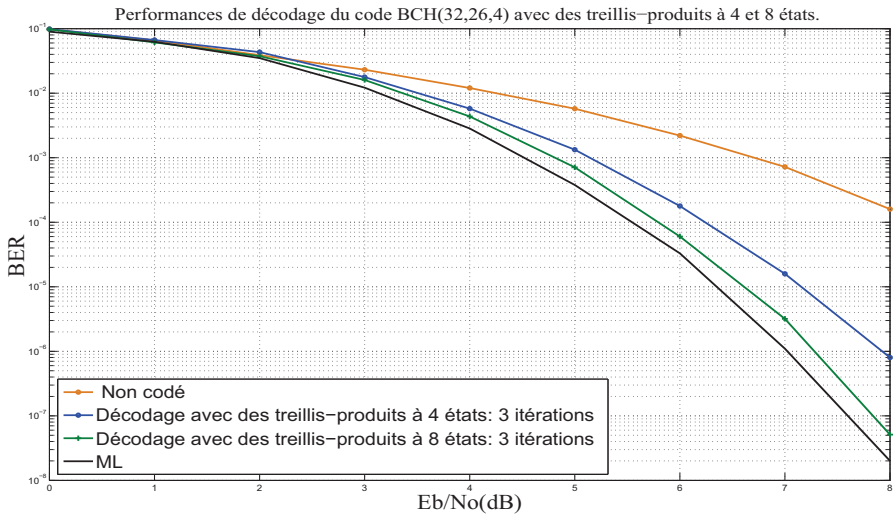

Figure 4.22. Performances de l'algorithme en fonction de la taille des treillis-produits pour le code BCH(32,26,4).

4.3.4.2 Sectionnalisation des treillis-produits

La sectionnalisation des treillis-produits permet de réduire le nombre de sections nulles dans ces treillis en les fusionnant avec d'autres sections non nulles. Les bits qui ne profitaient pas du décodage itératif vont en profiter dans le cadre des symboles qu'ils forment avec les autres bits car les treillis-produits sectionnalisés échangent des informations extrinsèques sur les symboles et plus sur les bits. La sectionnalisation permet aussi de réduire la complexité globale de décodage.

Dans ce paragraphe nous allons montrer l'apport de la sectionnalisation sur les performances de l'algorithme en présentant les résultats de simulation de décodage avec des treillis-produits sectionnalisés pour les deux codes de Hamming(8,4,4) et de Golay(24,12,8).

La Figure 4.23 compare les performances de l'algorithme obtenues avec des treillis-produits non sectionnalisés et celles obtenues avec des treillis-produits sectionnalisés pour le code de Hamming(8,4,4). Dans les deux cas, les treillis-produits sont à 4 états et dans le cas de sectionnalisation, les treillis sont sectionnalisés par groupe de deux sections. Nous constatons, en observant les courbes de taux d'erreur binaire sur cette figure que la sectionnalisation des treillis-produits améliore les performances de l'algorithme. En effet, à un taux d'erreur binaire 10^{-4} le décodage avec des treillis-produits sectionnalisés par groupe de 2 sections apporte un gain d'environ 0.5 dB par rapport au décodage avec des treillis-produits non sectionnalisés.

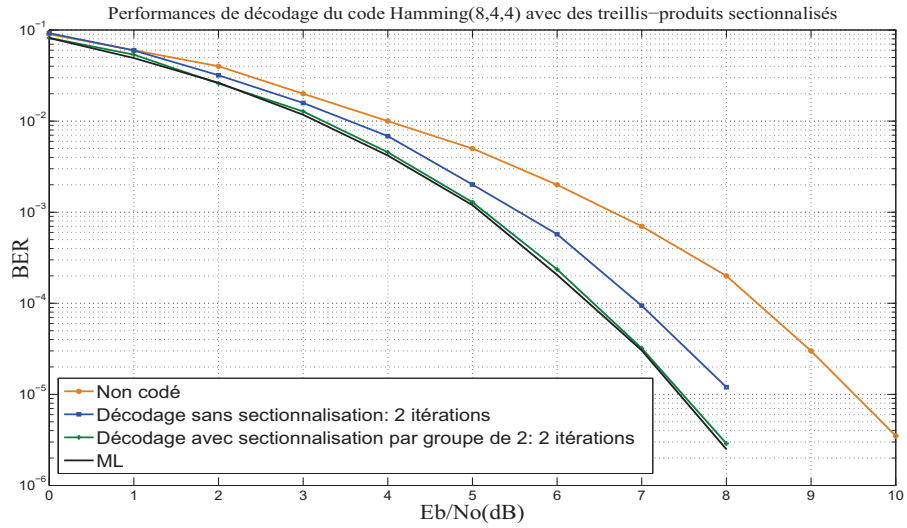

Figure 4.23. Performances de l'algorithme de dcodage avec des treillis-produits sectionnaliss pour le code de Hamming(8,4,4).

La Figure 4.24 compare les performances de l'algorithme obtenues avec des treillis-produits non sectionnalisés et celles obtenues avec des treillis-produits sectionnalisés pour le code de Golay(24,12,8). Dans les deux cas, les treillis-produits sont à 4 états et dans le cas de sectionnalisation, les treillis sont sectionnalisés par groupe de 2, 4 et 6 sections.

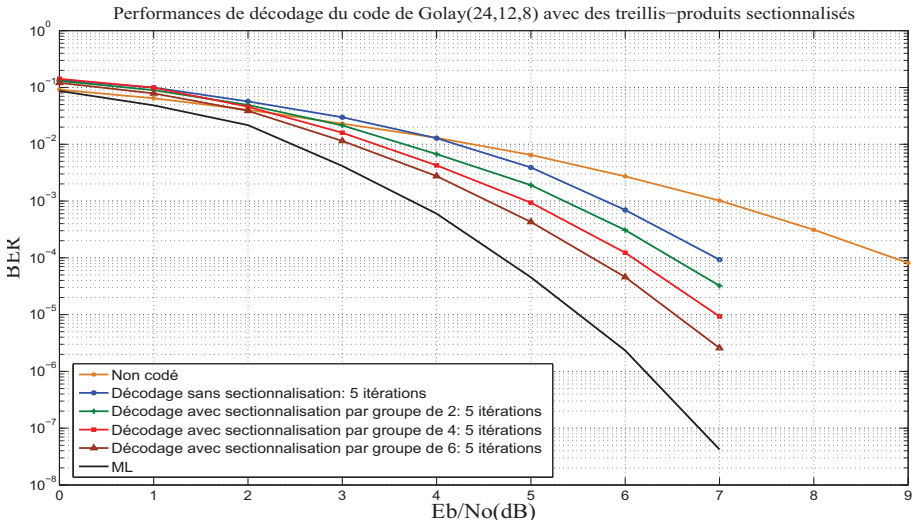

Figure 4.24. Performances de l'algorithme de dcodage avec des treillis-produits sectionnaliss pour le code de Golay(24,12,8).

En observant les courbes de taux d'erreur binaire sur la Figure 4.24 ci-dessus, nous pouvons faire deux constats. Le premier constat est que la sectionnalisation améliore les performances de l'algorithme. En effet, à un taux d'erreur binaire de 10^{-4}, le décodage avec des treillis-produits sectionnalisés par groupe de 6 apporte un gain d'environ 1.3 dB par rapport au décodage avec les mêmes treillis-produits non sectionnalisés. Le deuxième constat concerne la proportionnalité entre les performances et le nombre de sections groupées sur les treillis-produits. En effet, à un taux d'erreur binaire de 10^{-4}, le décodage avec des treillis-produits sectionnalisés par groupe de 4 apporte un gain d'environ 0.4 dB par rapport au décodage avec les mêmes treillis-produits sectionnalisés par groupe de 2 et le décodage avec des treillis-produits sectionnalisés

par groupe de 6 apporte un gain d'environ 0.5 dB par rapport au décodage avec des treillis-produits sectionnalisés par groupe de 4.

4.3.4.3 Décodage par deux matrices de contrôle différentes du code

Pour assurer une couverture équilibrée entre tous les bits, nous utilisons en même temps deux matrices de contrôle différentes dans le décodage. Dans ce cas, si un bit n'est pas couvert par les treillis-produits issus de l'une des deux matrices, il le sera par les treillis-produits issus de l'autre matrice. Tous les bits sont donc couverts et profitent du décodage itératif pour augmenter leur probabilités d'être décidés correctement. La deuxième matrice de contrôle doit être soigneusement choisie pour que ses treillis-produits ne contiennent pas des sections nulles dans les positions où les treillis-produits issus de la première matrice de contrôle contiennent des sections nulles.

Nous donnons un exemple simple du code de Hamming(8,4,4) dont une matrice de contrôle est définie par:

$$\boldsymbol{H} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.22}$$

Les 2 treillis-produits issus des lignes de cette matrice de contrôle contiennent chacun 2 sections nulles. Le premier treillis-produits formé par les deux premières lignes contient 2 sections nulles dans les positions 7 et 8. Le deuxième treillis-produits formé par les deux dernières lignes de la matrice contient 2 sections nulles dans les positions 5 et 6. Donc globalement nous avons 4/8 bits (bits 5, 6, 7 et 8) ne profitent pas du décodage itératif. L'idée présentée dans ce paragraphe est d'utiliser conjointement avec cette matrice une autre matrice de contrôle dont les treillis-produits couvrent ces 4 bits et donc leurs faire profiter du décodage itératif. Nous avons plusieurs matrices de contrôle qui peuvent assurer cette couverture. La matrice de contrôle ci-après semble un

bon choix:

$$\boldsymbol{H}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{4.23}$$

La Figure 4.25 compare les performances de l'algorithme obtenues avec la matrice de contrôle $\boldsymbol{H}$ seule et celles obtenues avec les deux matrices de contrôle $\boldsymbol{H}$ et $\boldsymbol{H}_1$ du code de Hamming(8,4,4). Les treillis-produits dans les deux cas de décodage sont formés par 2 lignes. Nous pouvons constater que les performances de décodage avec deux matrices de contrôle à 1 itération sont nettement supérieures de celles obtenues par le décodage avec une seule matrice de contrôle à 2 itérations. En effet, à un taux d'erreur binaire de 10^{-4} le premier cas de décodage apporte un gain d'environ 0.4 dB par rapport au deuxième cas sans engendrer une complexité supplémentaire.

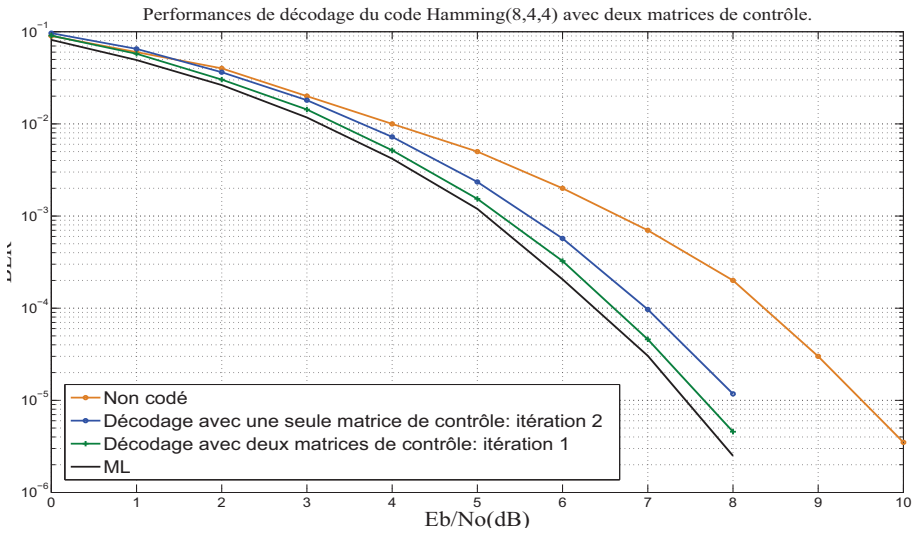

Figure 4.25. Performances de l'algorithme de dcodage avec deux matrices de contrle pour le code de Hamming(8,4,4).

La Figure 4.26 compare les performances de l'algorithme obtenues avec une seule matrice de contrôle et celles obtenues avec deux matrices de contrôle du code de

Golay(24,12,8). Les treillis-produits dans les deux cas de décodage sont formés par 4 lignes.

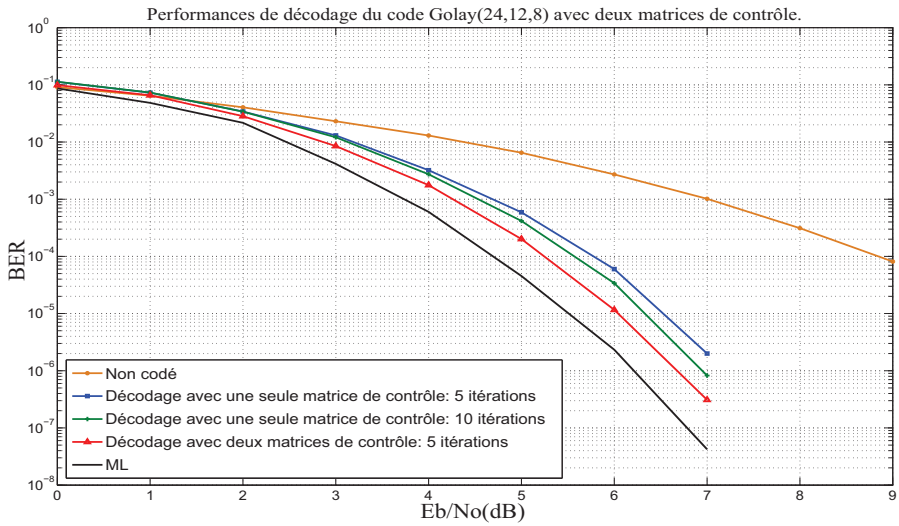

Figure 4.26. Performances de l'algorithme de dcodage avec deux matrices de contrle pour le code de Golay(24,12,8).

Nous constatons en observant les courbes de taux d'erreur binaire sur la Figure 4.26 que l'utilisation des deux matrices de contrôle conjointement dans le décodage améliore les performances de l'algorithme et accélère sa vitesse de convergence. En effet, à un taux d'erreur binaire de 10^{-5}, le décodage avec deux matrices de contrôle à 5 itérations apporte un gain d'environ 0.3 dB par rapport au décodage avec une seule matrice de contrôle à 10 itérations. Le premier cas de décodage apporte ce gain sans augmenter la complexité de décodage car il le réalise avec un nombre de treillis-produits 2 fois plus grand que dans le deuxième cas mais avec un nombre d'itérations 2 fois plus petit.

Dorénavant, nous retenons le décodage avec deux matrices de contrôle c'est-à-dire tous les résultats de simulations présentés ci-après sont basés sur un décodage avec deux matrice de contrôle du code sauf indication contraire.

Nous allons montrer que les performances de l'algorithme peuvent approcher celles du décodage optimal ML-exhaustif en jouant sur les deux paramètres n_L et n_S où n_L est

le nombre de lignes formant un treillis-produit et n_S est le nombre de sections groupées. Nous parlons donc d'une configuration de décodage (n_L, n_S).

La Figure 4.27 présente les performances de l'algorithme pour le décodage du code de Golay(24,12,8) avec une configuration ($n_L = 2$, $n_S = 6$).

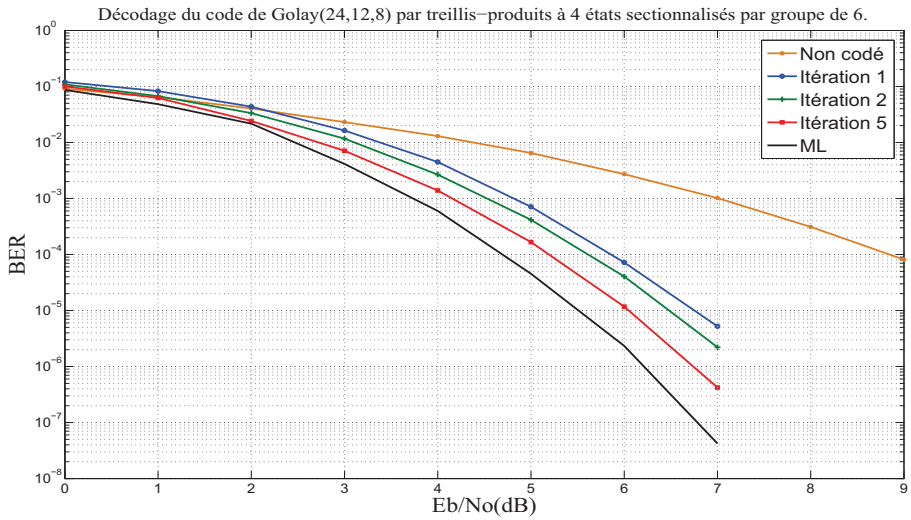

Figure 4.27. Performance de l'algorithme de dcodage avec des treillis-produits 4 tats sectionnaliss par groupe de 6 pour le code de Golay$(24, 12, 8)$.

La Figure 4.28 présente les performances de l'algorithme pour le décodage du code de Golay(24,12,8) avec une configuration ($n_L = 3$, $n_S = 6$).

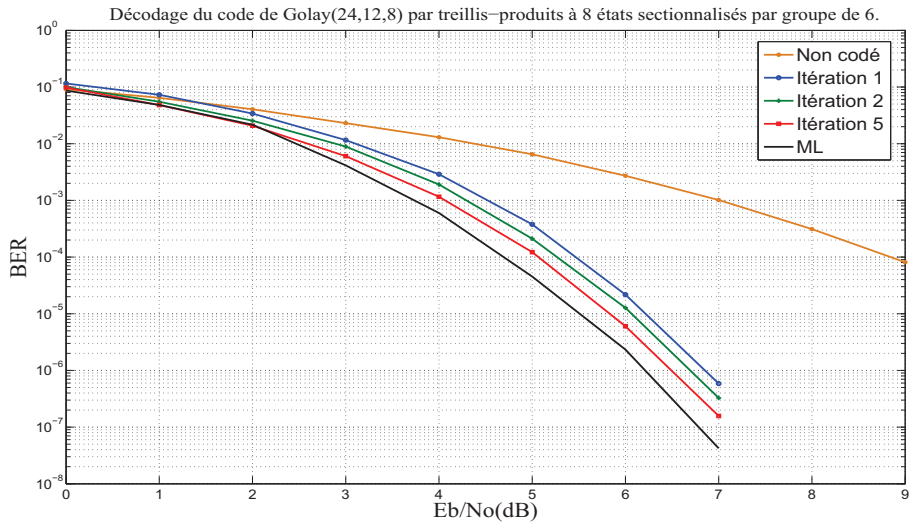

Figure 4.28. Performance de l'algorithme de dcodage avec des treillis-produits 8 tats sectionnaliss par groupe de 6 pour le code de Golay$(24,12,8)$.

La Figure 4.29 présente les performances de l'algorithme pour le décodage du code de Golay(24,12,8) avec une configuration ($n_L = 4$, $n_S = 6$).

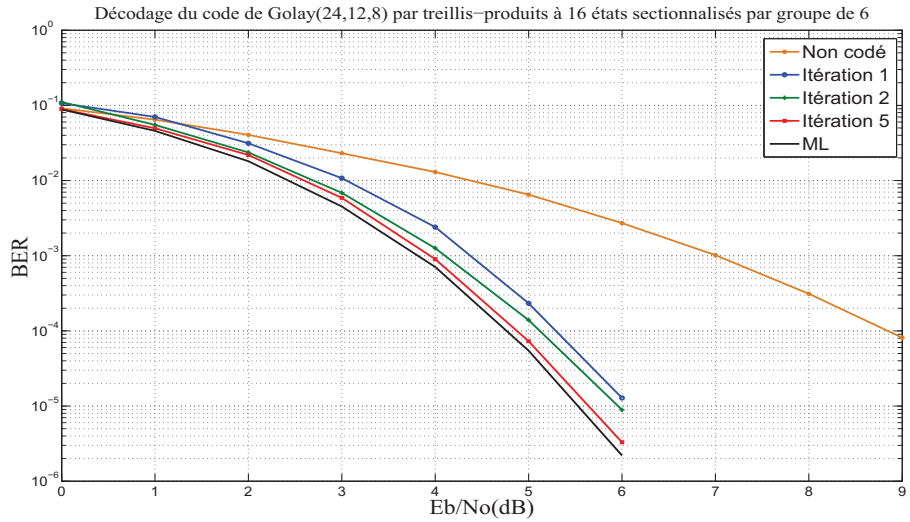

Figure 4.29. Performance de l'algorithme de dcodage avec des treillis-produits 16 tats sectionnaliss par groupe de 6 pour le code de Golay$(24,12,8)$.

La Figure 4.30 compare les performances des 3 configurations ($n_L = 2$, $n_S = 6$), ($n_L = 3$, $n_S = 6$) et ($n_L = 4$, $n_S = 6$) ci-dessus pour le décodage du code de Golay(24,12,8). Ces résultats confirment aussi que l'augmentation de la taille des treillis-produits améliore les performances de décodage. En effet, à un taux d'erreur binaire 10^{-5} le décodage avec des treillis-produits à 8 états apporte un gain d'environ 0.25 dB par rapport au décodage avec des treillis-produits à 4 états et le décodage avec des treillis-produits à 16 états apporte un gain d'environ 0.2 dB par rapport au décodage avec des treillis-produits à 8 états.

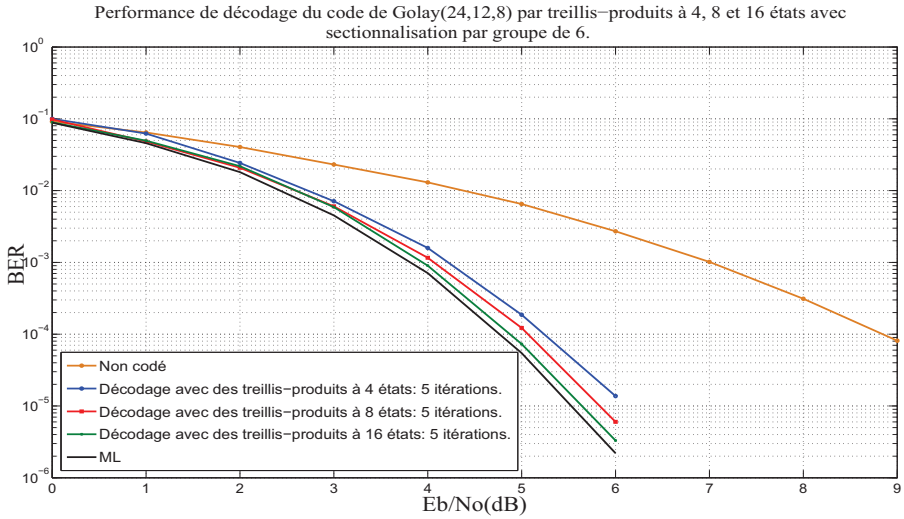

Figure 4.30. Performance de l'algorithme de dcodage avec des treillis-produits 4, 8 et 16 tats sectionnaliss par groupe de 6 pour le code de Golay$(24,12,8)$.

Le choix de la configuration de décodage (n_L, n_S) est contraint par le compromis entre les performances et la complexité. Certes, l'augmentation de la taille des treillis-produits améliore les performances mais elle augmente aussi la complexité de décodage. Donc, il y a toujours un compromis à faire selon le besoin en termes de performances et les contraintes de complexité imposées par l'application.

Le tableau 4.1 donne les performances pour quelques configurations de décodage (n_L, n_S)

pour le code de Golay$(24,12,8)$. La valeur numérique associée à chaque configuration est l'écart de performance (en dB) calculé par rapport au ML (Maximum Likelihood) pour un taux d'erreur binaire de 10^{-4} et 3 itérations.

n_L \ n_S	2	4	6
2 (12 treillis)	1.1 dB	0.8 dB	0.6 dB
3 (8 treillis)	0.8 dB	0.6 dB	0.4 dB
4 (6 treillis)	0.6 dB	0.4 dB	0.2 dB

Table 4.1. Comparaison entre quelques configurations de dcodage (n_L, n_S) pour le code Golay$(24,12,8)$.

4.3.5 Etude de la complexité de décodage

La complexité de la méthode de décodage proposée est la complexité d'un décodeur élémentaire multipliée par le nombre total de ces décodeurs n_p et le nombre maximal d'itérations I. Un décodeur élémentaire effectue un algorithme BCJR sur un treillis-produit sectionnalisé. Considérons une section t de ce treillis-produit. Le décodage commence par une phase de prétraitement dans laquelle, une table de correspondance U_{γ_t} est calculée. Cette table contient les probabilités, $\gamma_t(s',s)$, des branches simples distinctes et les probabilités, $\Gamma_t(s',s)$, des branches multiples distinctes. Le calcul de cette table passe par les étapes suivantes:

1. Calcul des probabilités $\gamma_t(s',s)$ pour toutes les simples branches distinctes de la section t, en utilisant les équations 4.10 et 4.13, ceci demande $N_b^t.N_d^t.(m_t-1)$ additions et $N_b^t.N_d^t$ multiplications dans la première itération du décodage et $N_b^t.N_d^t$ multiplications dans les autres itérations.

2. Calcul des probabilités des branches multiples distinctes en utilisant l'équation 4.14, ceci demande $N_d^t(N_b^t - 1)$ additions. Dans le cas où le treillis ne comporte pas des branches multiples (*i.e.* $N_b^t = 1$), nous avons besoin de calculer les probabilités $\gamma_t(s', s)$ seulement.

Donc, le calcul de la table U_{γ_t} demande un total de $N_m^t(\gamma)$ multiplications et $N_a^t(\gamma)$ additions où $N_m^t(\gamma)$ et $N_a^t(\gamma)$ sont donnés par:

- Dans la première itération:

$$N_a^t(\gamma) = \begin{cases} N_d^t.N_b^t.(m_t - 1) + N_d^t(N_b^t - 1) & \text{si } N_b^t > 1 \\ N_d^t.(m_t - 1) & \text{si } N_b^t = 1 \end{cases} \tag{4.24}$$

$$N_m^t(\gamma) = N_b^t.N_d^t \tag{4.25}$$

- Dans les autres itérations:

$$N_a^t(\gamma) = \begin{cases} N_d^t(N_b^t - 1) & \text{si } N_b^t > 1 \\ N_d^t & \text{si } N_b^t = 1 \end{cases} \tag{4.26}$$

$$N_m^t(\gamma) = N_b^t.N_d^t \tag{4.27}$$

Après la phase du calcul de la table U_{γ_t}, l'algorithme BCJR effectue ses deux phases forward et backward pour calculer les probabilités α et β. Le calcul des probabilités α, à l'aide de équation 1.47 et en utilisant l'expression de la probabilité de branche multiple dans l'équation 4.14, demande $N_a^t(\alpha)$ additions et $N_m^t(\alpha)$ multiplications où $N_a^t(\alpha)$ et $N_m^t(\alpha)$ sont donnés, respectivement, par:

$$\begin{aligned} N_a^t(\alpha) &= (deg(s_t)_{in} - 1).n_{s_t} \\ &= N_B^t - n_s^t \end{aligned} \tag{4.28}$$

$$N_m^t(\alpha) = N_B^t \tag{4.29}$$

Le calcul des probabilités β à l'aide de équation 1.48 et en utilisant l'expression de la probabilité de branche multiple dans l'équation 4.14 demande $N_a^t(\beta)$ additions et $N_m^t(\beta)$ multiplications où $N_a^t(\beta)$ et $N_m^t(\beta)$ sont donnés, respectivement, par:

$$\begin{aligned} N_a^t(\beta) &= (deg(s_t)_{out} - 1).n_s^{t-1} \\ &= N_B^t - n_s^{t-1} \end{aligned} \tag{4.30}$$

$$N_m^t(\beta) = N_B^t \tag{4.31}$$

Après les deux phases forward et backward et les calculs des probabilités α et β, l'algorithme BCJR calcule les LLR extrinsèque pour chaque symbole de la section t. Les calculs des LLRs extrinsèques L_e^d et LLRs *a priori* L_a^d des symboles effectués, respectivement, par les équations 4.17 et 4.18 demandent $(N_B^t - 1) + n_p - 1$ additions et N_B^t multiplications. Donc, la complexité de décodage pour un treillis-produit T_p est $N_a(T_p)$ additions et $N_m(T_p)$ multiplications telles que:

$$N_a(T_p) = \sum_{t=0}^{L-1} (N_a^t(\gamma) + N_a^t(\alpha) + N_a^t(\beta) + N_B^t + n_p - 2) \tag{4.32}$$

$$N_m(T_p) = \sum_{t=0}^{L-1} (N_m^t(\gamma) + N_m^t(\alpha) + N_m^t(\beta) + N_B^t) \tag{4.33}$$

où n_p est le nombre des treillis-produits et L est la longueur d'un treillis-produit.

Dans la phase finale, l'algorithme calcul les LLRs a posteriori des symboles à l'aide de l'équation 4.20, ceci demande $L2^{m_t} n_p$ additions.

La complexité de décodage peut être réduite, sans dégrader les performances, en ne faisant pas les calculs pendant les phases forward/backward de l'algorithme BCJR sur les sections nulles du treillis-produit. Pour donner un ordre de grandeur de la complexité de cette méthode, nous calculons sa complexité pour le code de Golay$(24, 12, 8)$. Le décodage de ce code est effectué à l'aide de deux matrices de contrôles et avec des treillis-produits à 16 états ($n_L = 4$) et sectionnalisés par groupe de 6 sections ($n_S = 6$). Le nombre des treillis-produits est $n_p = 6$. Chaque treillis-produit est donc constitué de 4 sections. La 1$^{\text{ère}}$ et 4$^{\text{ème}}$ section de ce treillis-produit, si elles ne sont pas nulles,

sont constituées chacune de $N_B^t = 16$ branches multiples, $t = 0,3$. Chaque branche multiple est constituée de $N_b^t = 4$ branches simples, $t = 0,3$. La $2^{\text{ème}}$ et $3^{\text{ème}}$ section de ce treillis-produit, si elles ne sont pas nulles, sont constituées chacune de $N_B^t = 256$ branches multiples, $t = 1,2$. Chaque branche multiple est constituée de $N_b^t = 4$ branches simples, $t = 1,2$. Le nombre de branches multiples distinctes par section est égal à $N_d^t = 16$ branches, $t = 0,1,2,3$. Dans la phase forward sur la $1^{\text{ère}}$ section, aucun calcul n'est nécessaire car pour chaque état d'arrivée, il y a une seule branche multiple qui y arrive et donc la probabilité de cet état est égale à celle de cette branche multiple qui est déjà stockée dans la table de correspondance U_{γ_t}. Pareil aussi pour la phase backward sur la $4^{\text{ème}}$ section. La complexité du treillis-produit est donc la complexité de ses deux sections au milieu. Il suffit donc de calculer la complexité d'une de ces dernières pour en déduire la complexité totale du treillis-produit. Le calcul de la table de correspondance U_{γ_t} demande un total de $N_m^t(\gamma) = 4 \times 16 = 64$ multiplications et $N_a^t(\gamma) = 16 \times 4 \times (6-1) + 16(4-1) = 338$ additions dans la première itération et $N_a^t(\gamma) = 16 \times 4 + 16(4-1) = 112$ multiplications dans les autres itérations. Le calcul de α demande $N_a^t(\alpha) = N_B^t - n_s^t = 256 - 16 = 240$ additions et $N_m^t(\alpha) = N_B^t = 256$ multiplications. De même, le calcul de β demande $N_a^t(\beta) = 240$ additions et $N_m^t(\beta) = 256$ multiplications. Les calculs des LLRs extrinsèques L_e et LLRs *a priori* L_a^d pour les symboles demandent $(256-1)+6-1 = 260$ additions et 256 multiplications. Donc, la complexité globale d'une section complète, est 768 multiplications et 740 additions. Parmis les 6 treillis-produit il y a 4 treillis qui contiennent chacun une section nulle. La complexité d'un treillis-produit contenant une section nulle est 768 multiplications et 740 additions. Pour un treillis-produit qui ne contient pas de section nulle, sa complexité est 1562 multiplications et 1480 additions. Le calcul des LLRs a posteriori des symboles demande $4 \times 2^6 \times \times 6 = 1536$ additions.

La complexité globale de décodage est donnée dans les tableaux 4.2 et 4.3 pour quelques configurations de décodage (n_L, n_S).

Algorithme de décodage	opérations $(+,\times)$
1 itération	8 592
2 itérations	15 424
5 itérations	35 920
Algorithme de décodage par liste de Adde [74]	885
algorithme ML-exhaustif	98 304

Table 4.2. Complexits de dcodage pour quelques itrations avec une configuration de dcodage $(n_L = 3, n_S = 6)$ pour le code Golay$(24,12,8)$.

Algorithme de décodage	opérations $(+,\times)$
1 itération	15 206
2 itérations	28 304
3 itérations	40 868
Algorithme de décodage par liste de Adde [74]	885
algorithme ML-exhaustif	98 304

Table 4.3. Complexits de dcodage pour quelques itrations avec une configuration de dcodage $(n_L = 4, n_S = 6)$ pour le code Golay$(24,12,8)$.

La Figure 4.31 donne une comparaison entre les performances de l'algorithme avec les deux configurations de décodage $(n_L = 3, n_S = 6)$ et $(n_L = 4, n_S = 6)$ et celles de l'algorithme par liste proposé par Adde *et al* [74] pour le code de Golay(24,12,8). Les deux algorithmes donnent des performances quasi-similaires mais notre algorithme est beaucoup plus complexe. Ce dernier peut présenter un avantage sur l'algorithme par liste pour le décodage des codes en bloc de longueurs supérieures à 48. En effet, la

complexité de l'algorithme de décodage par liste s'explose pour des codes de longueurs supérieure à 48 alors que celle de notre algorithme est linéaire avec la complexité des treillis-produits qui est réduite.

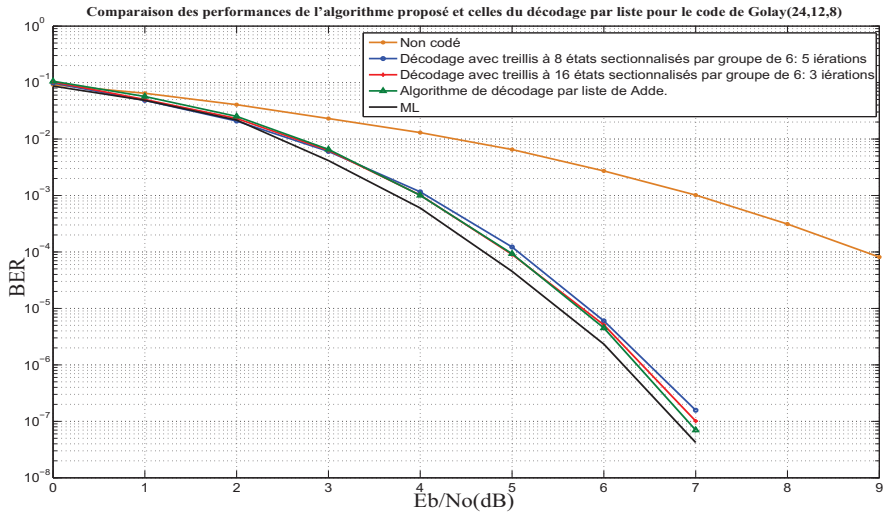

Figure 4.31. Comparaison de performances de l'algorithme propos avec les deux configurations de dcodage $(n_L = 3, n_S = 6)$ **et** $(n_L = 4, n_S = 6)$ **et celles du dcodage par liste [74] pour le code de Golay**$(24, 12, 8)$.

4.3.6 Implantation matérielle du décodeur pour le code de Golay$(24, 12, 8)$

Dans cette section nous présentons une architecture matérielle permettant d'implanter ce décodeur sur une carte FPGA pour le code de Golay$(24, 12, 8)$. Le travail présenté dans cette section a été fait en collaboration avec Anaïs HAMON et fatou SAMBOU étudiantes à Télécom Bretagne dans le cadre de leur projet "FIP 3A". Cette architecture matérielle est conçue pour une structure de décodage constituée par des treillis-produits à 4 états. Pour des raisons de complexité, nous avons utilisé l'algorithme Max-Log-MAP pour le décodage sur les treillis-produits.

L'architecture proposée est composée de deux parties:

1. Une partie opérative constituée des différents éléments qui vont permettre de

faire les différents calculs et comparaisons exigés localement par l'algorithme Max-Log-MAP ou globalement dans le processus itératif. Ces éléments prévoient aussi l'espace mémoire nécessaire pour stocker certaines métriques calculées par l'algorithme Max-Log-MAP.

2. Une partie de contrôle qui permettra de gérer la partie temporelle en donnant les instructions aux différents éléments de la partie opérative.

4.3.6.1 Partie opérative

Les fonctions réalisées dans le cadre de l'algorithme sont représentées par des blocs. Un bloc est défini par ses entrées, ses sorties et les tâches qu'il effectue. Les schémas de blocs décrits par les figures 4.32 et 4.33 permettent de déterminer les différents blocs qui sont nécessaires à l'implémentation du décodeur.

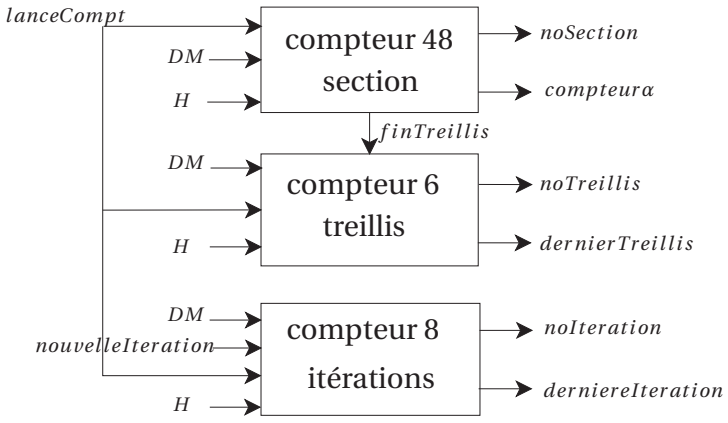

Figure 4.32. Blocs des compteurs.

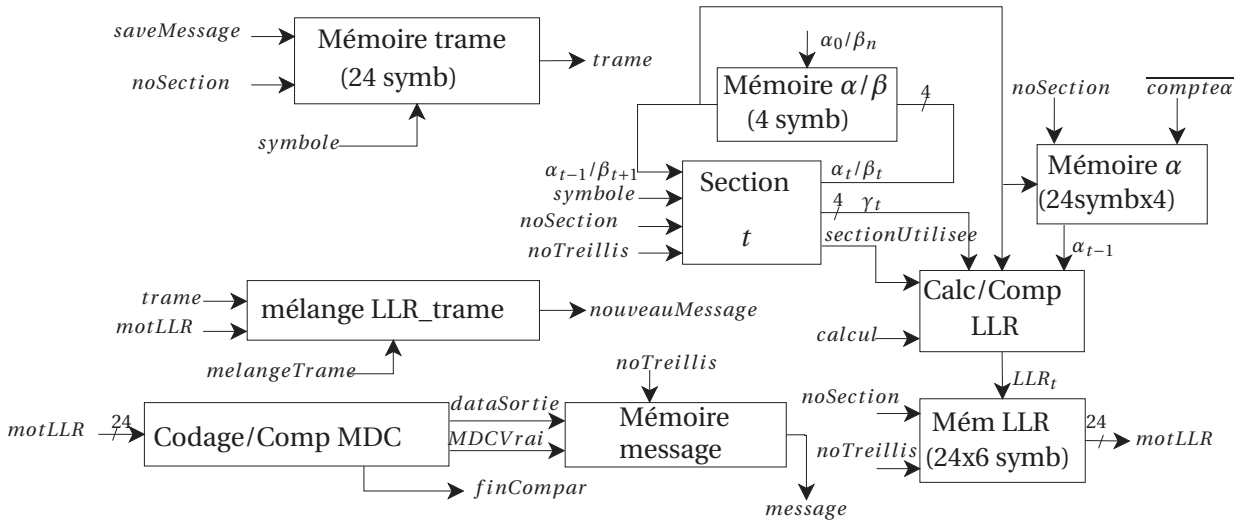

Figure 4.33. Schma blocs de la partie oprative.

Les détails des différents blocs dans les schémas précédents ainsi que leurs fonctionnalités sont mis en annexe *B*.

4.3.6.2 Partie contrôle

La partie contrôle permet de gérer temporellement le déroulement des étapes du décodage. Elle peut être représentée par une machine d'états en se basant sur la partie opérationnelle déjà conçue.

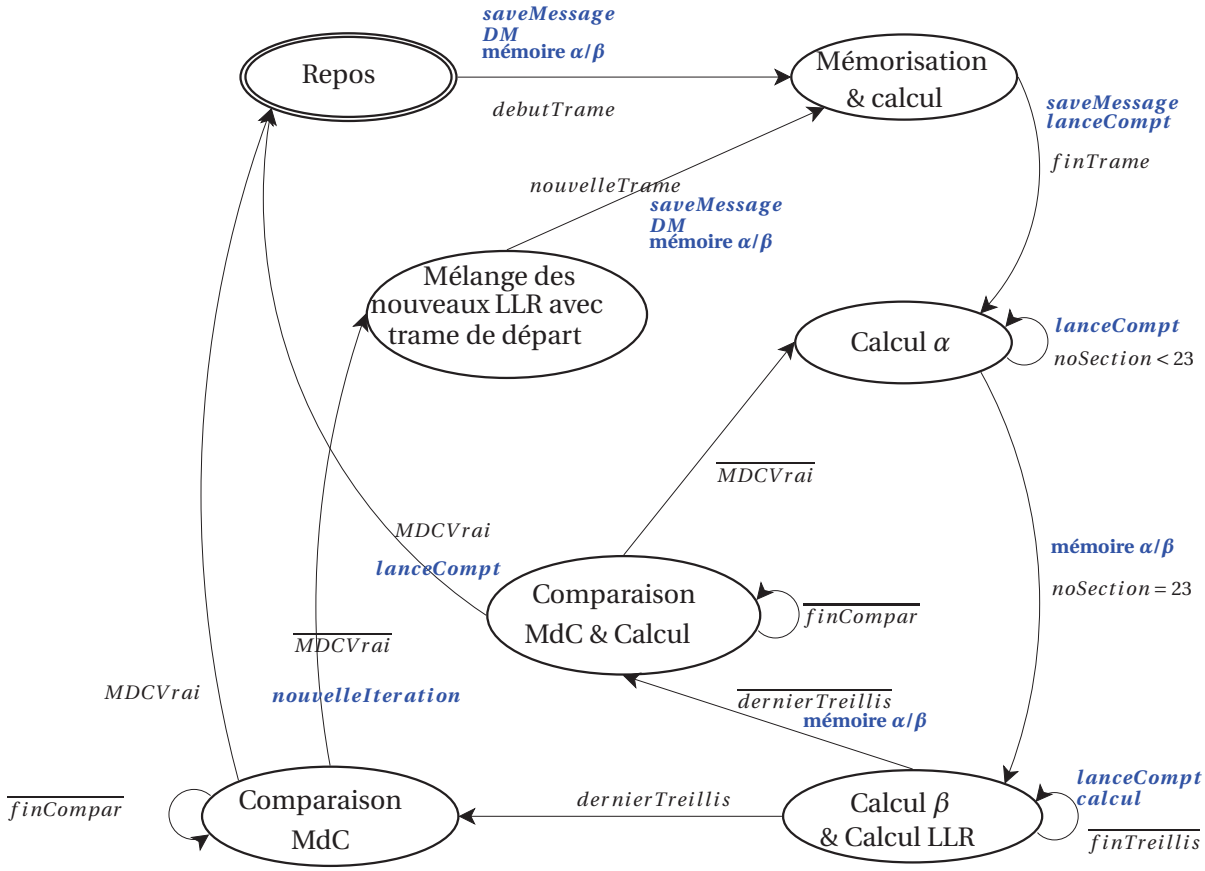

Figure 4.34. Partie contrle.

Chaque état pilote plusieurs tâches à effectuer par les diffréents blocs dans la partie opérative à travers les signaux marqués en gras sur la Figure 4.34. Les signaux marqués en noir sur la même figure servent à véhiculer des acquittements à l'état de la part des blocs, ceci permet de faire la transition d'un état à l'autre. La machine d'état est synchrone, les changements se produisent donc au coup d'horloge suivant l'arrivée du signal.

4.3.6.3 Implantation en VHDL sur FPGA

L'implantation réalisée se base sur le schéma des blocs et la machine d'états présentés précédemment. La synthèse des blocs permettra d'obtenir la surface que celui-ci va

prendre sur le silicium et donc le nombre de portes nand2 équivalentes. Le tableau ci-dessous donne une synthèse du nombre de portes nand2 pour chacun des blocs précédent.

Bloc (surface en μm^2)	**Complexité (nombre équivalent de portes nand2 de $2.82\mu m^2$)**
Compteur 6 (105.13)	38
Compteur 8 (88.20)	32
Compteur 48 (251.90)	90
Mémoire α/β (812.85)	289
Automate (287.18)	102
Section t (9670.95)	3 430
Codage/Comp MDC (13 697.81)	4 858
Mémoire α (26860.78)	9 525
Calc/Comp LLR (5004.12)	1 775
Mémoire LLR	-
Mémoire trame	-
Mémoire message	-
mélange LLR_trame	-
Total	20 193

Table 4.4. Complexit matrielle de l'algorithme de dcodage pour le code de Golay$(24,12,8)$

4.4 Conclusion du chapitre 4

Dans ce chapitre nous avons présenté deux algorithmes de décodage itératif pour les codes en bloc basés sur des treillis-produits. Les deux algorithmes utilisent des treillis-produits de complexité réduite construits à partir des treillis élémentaires représentant des lignes de la matrice de contrôle du code.

Le premier algorithme fait une recherche du mot le plus probable sur chacun des treillis-produits au lieu de faire la recherche sur le treillis global du code. Il utilise l'algorithme List-Viterbi pour faire la recherche sur chaque treillis-produit afin de constituer une liste des mots candidats dans laquelle le mot à distance euclidienne minimale du mot reçu est sélectionné comme décision. Les performances, en termes de taux d'erreur binaire, de cet algorithme ont été ensuite présentées pour les trois codes de Hamming$(8,4,4)$, BCH$(31,26,3)$ et BCH$(32,26,4)$. Ces performances ont été obtenues avec des treillis-produits à 4 états.

Le deuxième algorithme est un algorithme SISO qui fait coopérer des treillis-produits dans un processus itératif pour finalement donner une estimation des LLRs *a posteriori* des symboles reçus. Nous avons ensuite présenté ses performances, en termes de taux d'erreur binaire, pour plusieurs codes en bloc. La vitesse de convergence de l'algorithme est contrainte par la présence des sections nulles dans les treillis-produits. Nous avons présenté ensuite 3 techniques permettant de limiter l'effet de ces sections nulles et d'améliorer les performances. A la fin, nous avons montré que la combinaison de ces 3 techniques permet à l'algorithme d'atteindre les performances quasi-optimales avec une complexité raisonnable.

CONCLUSION GÉNÉRALE

L'objectif principal de cette thèse était de contribuer au décodage des codes en bloc linéaires courts. Nos travaux se sont orientés, dans un premier temps, vers le décodage d'une classe particulière des codes en bloc appelés codes Cortex. L'objectif était de proposer des algorithmes de décodage en exploitant la structure concaténée des codes Cortex afin d'assurer une complexité linéaire du décodage avec la taille des codes. Dans un second temps nous avons élargi nos recherches à des algorithmes de décodage très généraux valables pour tout code en bloc.

Décodage des codes Cortex (chapitre 2)

La première idée de décodage pour les codes Cortex s'imposait naturellement par leur structure concaténée. Il s'agit d'un décodage itératif de type Belief Propagation entre les différents codes de base constituant cette structure. Le problème qui se pose toujours pour ce type de décodage des codes Cortex est l'indisponibilité des informations sur les variables internes ou cachées dans la structure Cortex et la difficulté de leur estimation. Nous avons ensuite proposé une méthode permettant d'estimer ces variables cachées en exploitant une propriété des mots de code de base de Hamming(8,4,4) qui fait que les 4 bits de redondance (respectivement bits d'information) se déduisent très simplement des 4 bits d'information (respectivement bits de redondance) à l'aide d'un bit d'inversion calculé par la somme, modulo 2, de ces 4 bits. Malgré l'estimation des variables internes, ce décodage itératif est confronté aux cycles dans la structure qui ramènent des informations aux codes de bases dont ils étaient origines et créent donc une corrélation qui dégrade les performances du décodage. Les performances de cet algorithme ont été évaluées par simulations pour le code de Golay(24,12,8) construit sous forme Cortex. L'algorithme ne donne malheureusement pas de bonnes perfor-

mances. Mais nous avons appris de cette tentative que pour ce type de décodage, qui n'est pas compatible avec la présence des cycles, il faudrait penser à des modifications astucieuses et profondes sur l'algorithme BP qui le permettent de transformer ces cycles en un atout au lieu d'un handicap.

La deuxième méthode de décodage est aussi itérative. Elle consiste à réduire la structure Cortex aux codes de base de l'étage central et à remplacer les autres codes de base par des équations booléennes. Chaque code de base de l'étage central constitue un treillis avec les équations booléennes. Les treillis ainsi constitués vont ensuite coopérer dans un processus itératif en échangeant entre eux des informations extrinsèques sur les bits en commun. L'avantage de cette méthode est qu'elle ne nécessite pas d'estimation des variables internes mais la structure itérative constituée par les treillis peut être confrontée aux cycles courts qui font retourner à chaque treillis des informations qu'il a déjà calculées dans une itération précédente. En effet, pour le code de Golay(24,12,8) construit avec trois étages de code de Hamming(8,4,4), la structure itérative, composée de 3 treillis, a une forme triangulaire c'est-à-dire qu'à la troisième itération de décodage chaque treillis reçoit des informations qu'il a déjà produites à la première itération. Nous posons ci-dessous certaines questions qui peuvent représenter des pistes de recherche pour améliorer les performances de cette méthode de décodage:

- Que se passe-t-il si l'étage central de la structure Cortex contient plus de 3 codes de base? Cela va certainement augmenter la longueur du cycle dans la structure de décodage et donc va améliorer les performances de l'algorithme.
- La présence des cycles dans la structure de décodage est imposée par le fait que les treillis n'ont pas les même bits en commun. Si les treillis ont les mêmes bits en commun, la structure aura la forme d'un arbre et donc sans cycles. Cela améliorera-il les performances de décodage? Et que se passe-il si de plus tous les bits du mot de code sont représentés sur chaque treillis? Cela va certainement améliorer les performances car dans ce cas chaque bit est couvert par tous les treillis et

donc profite davantage du décodage itératif mais les performances peuvent-elles approcher les performances optimales?

- La taille des treillis (nombre d'états) est imposée par la dimension des codes de base. Dans le cas du code de Golay(24,12,8), construit avec trois étages de code de Hamming(8,4,4), les treillis ont 16 états car la dimension du code de base est 4. Que se passe-il si les codes de base utilisés sont des codes de Hadamard$(4,2,2)$ conduisant ensuite à des treillis à 4 états? Ceci va certainement réduire la complexité globale de décodage.
- Dans notre cas nous avons utilisé l'algorithme sous-optimal Max-Log-MAP comme un algorithme SISO sur les treillis. Les performances peuvent être améliorées en utilisant un algorithme de décodage optimal comme l'algorithme BCJR. Ceci peut être utile surtout dans le cas où les treillis sont à 4 états c'est-à-dire de complexité réduite.

Décodage général des codes en bloc linéaires (chapitres 3 et 4)

Dans le chapitre 3 nous avons proposé un algorithme de décodage qui fait interagir, à l'aide d'un produit cartésien, des treillis élémentaires 2 à 2 issus de la matrice génératrice ou de contrôle du code sur plusieurs étapes afin de calculer des estimations des probabilités *a posteriori* des symboles reçus. Cet algorithme utilise une nouvelle approche consistant à calculer le résultat des interactions entre les treillis élémentaires par leurs probabilités *a posteriori* d'états et non pas par les probabilités extrinsèques sur les bits de code. A la fin de chaque interaction entre deux treillis élémentaires, l'algorithme calcule les probabilités *a posteriori* d'états de chacun de ces treillis par marginalisation des probabilités d'états du treillis-produit. Les performances de l'algorithme sont ensuite présentées pour 3 codes en bloc: Hamming$(8,4,4)$, Golay$(24,12,8)$ et QR$(48,24,12)$. L'algorithme donne de bonnes performances sur un canal binaire à effacement ou BEC pour les deux codes de Hamming(8,4,4) et Golay(24,12,8) en corrigeant parfaitement

tous les motifs d'au plus $(d_{min}-1)$ effacements mais il ne donne malheureusement pas de bon résultats pour le code QR(48,24,12). Il ne donne pas non plus de bonnes performances sur un canal gaussien ou AWGN. Ces mauvaises performances sont induites par la présence des sections nulles dans les treillis-produits. Une section nulle S_t dans un treillis est une section où chaque états s de l'étage t est connecté à un état homologue s de l'étage $t+1$ par l'ensemble des branches possibles. Deux états de cette section ne sont pas connectés entre eux s'ils sont différents. Pour réduire le nombre de sections nulles dans les treillis-produits, nous donnons ci-dessous deux propositions comme perspectives de cet algorithme de décodage:

- La première proposition consiste à faire une sectionnalisation des treillis-produits en fusionnant les sections nulles avec d'autres sections non-nulles pour former une seule section.
- La deuxième proposition consiste à augmenter le nombre de treillis élémentaires constituant un treillis-produit. Dans notre cas, les treillis-produits sont constitués chacun par le produit de 2 treillis élémentaires. Nous pouvons utiliser plus de 2 treillis élémentaires (par exemple 4) car cela réduit le nombre de sections nulles dans le treillis résultant et éventuellement limiter leurs effets négatifs sur les performances.

Dans le chapitre 4, nous avons proposé deux algorithmes de décodage pour les codes en bloc. Ces deux algorithmes utilisent des treillis-produits de complexité réduite construits à partir des treillis élémentaires représentant des lignes de la matrice de contrôle du code.

Le premier algorithme est un algorithme qui fait une recherche du mot le plus probable sur chacun des treillis-produits au lieu de faire la recherche sur le treillis global du code. Il utilise l'algorithme List-Viterbi sur chacun des treillis-produits pour effectuer la recherche. Les performances de cet algorithme ont été évaluées pour les codes de

Hamming(8,4,4), BCH(31,26,3) et BCH(32,26,4). L'algorithme donne de bonnes performances mais sa vitesse de convergence est inversement proportionnelle à la longueur du code. En effet, pour le code BCH(31,26,3), les performances de l'algorithme sont très proches de celles du décodage ML-exhaustif en effectuant seulement 5 itérations alors que pour le code BCH(32,26,4) il faut 7 itérations pour approcher les performances optimales. Ceci est justifié par le fait que l'augmentation de la longueur du code augmente le nombre de chemins dans les treillis-produits et donc rend plus difficile la recherche du chemin le plus probable. Pour accélérer la vitesse de convergence de l'algorithme, nous proposons d'augmenter la taille des treillis-produits en augmentant le nombre des lignes qui les constituent car cela diminue le nombre de chemins dans ce treillis-produit et facilite la recherche du chemin le plus probable.

Le deuxième algorithme est un algorithme SISO qui estime les logarithmes des rapports de vraisemblance ou LLRs (Log-Likelihood Ratio) *a posteriori* des symboles reçus. L'algorithme fait coopérer les treillis-produits dans un processus itératif dans lequel ils échangent des informations extrinsèques sur les symboles formant les étiquettes des branches de leurs sections. Ces informations extrinsèques sont calculées sur chaque treillis-produit en utilisant un algorithme SISO comme l'algorithme BCJR. Cet algorithme donne de bonnes performances pour certains codes mais pour d'autres codes sa vitesse de convergence est ralentie par la présence des sections nulles dans les treillis-produits qui empêchent certains symboles de profiter du décodage itératif. Nous nous sommes intéressés ensuite à ce problème posé par les sections nulles afin d'en trouver des solutions. Nous avons donc abouti à deux solutions. La première solution consiste à utiliser deux matrices de contrôle du code conjointement afin d'assurer une couverture des symboles présents dans ces sections nulles. La deuxième solution consiste à faire une sectionnalisation des treillis-produits en fusionnant les sections nulles avec d'autres sections non-nulles pour former une seule section. Ces solutions ont été efficaces pour limiter l'effet des sections nulles et accélérer la convergence de l'algorithme. En effet, pour le décodage du code de Golay(24,12,8), l'utilisation de ces deux solu-

tions permet d'apporter un gain jusqu'à 2 dB par rapport au décodage avec la matrice de contrôle seule et avec un nombre d'itérations deux fois plus petit et en atteignant les performances optimales du décodage ML. Une architecture matérielle permettant d'implanter ce décodeur sur une carte FPGA pour le code de Golay$(24, 12, 8)$ a été présentée dans le chapitre 4.

Nous donnons ci-dessous quelques propositions pour les travaux futurs sur cet algorithme de décodage:

- La sectionnalisation optimale des treillis-produits qui conduit à un nombre minimal d'opérations arithmétiques et d'allocations mémoire. Nous pouvons utiliser des algorithmes, déjà publiés, permettant de trouver la sectionnalisation optimale [85][86] ou concevoir des nouveaux algorithmes qui tiennent compte aussi de la nécessité de réduire les sections nulles dans ces treillis-produits.
- L'utilisation de plus de 2 matrices de contrôle dans le décodage afin d'assurer une couverture plus large des symboles et donc probablement améliorer les performances de décodage? Ceci est très utile en particulier pour le décodage des codes à matrices creuses comme les codes LDPC.
- Appliquer un coefficient de pondération sur les extrinsèques c'est-à-dire qu'à la fin de chaque itération, les extrinsèques de la minorité, suivant le signe, sont pondérées par un coefficient inférieur à 1. Ceci peut améliorer les performances car il attenue les contributions des extrinsèques les moins fiables et donc rend l'extrinsèque globale sur le symbole plus pertinente.

RÉFÉRENCES

[1] A. Osseiran, J.F. Monserrat, W. Mohr, *Mobile and Wireless Communications for IMT-Advanced and Beyond*, Wiley, August 2011.

[2] Internet Society, *Internet Society Global Internet report 2014*, 2014.

[3] Cisco, *Connections Counter: The Internet of Everything in Motion*, consulter la page: `http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342`

[4] C. Shannon, *A Mathematical Theory of Communication*, Bell Syst. Tech. Journal, vol. 27, pp. 623-656, 1948.

[5] R. Hamming, *Error detecting and error correcting codes*, Bell Syst. Tech. Journal, vol. 29, pp. 41-56, 1950.

[6] IEEE Information Theory Society, *Claude E. Shannon Award*, consulter la page: `http://www.itsoc.org/honors/claude-e-shannon-award`

[7] IEEE, *IEEE Richard W. Hamming Medal*, consulter la page: `http://www.ieee.org/about/awards/medals/hamming.html`

[8] D. Muller, *Application of Boolean Switching Algebra to Switching Cicuit Design*, IEEE Transaction on Computers, vol. 3, pp. 6-12, September 1954.

[9] I. Reed, *A Class of Multiple-Error-Correcting Codes and a Decoding Scheme*, IEEE Transaction on Information Theory, vol. 4, pp. 38-49, September 1954.

[10] P. Elias, *Coding for two noisy channels*, The 3rd London Symposium, Buttersworth's Scientific Publications, pp. 61-67, September 1955.

[11] P. Elias, *Error-free coding*, IRE Transaction on Information Theory, vol. 4, no. 4, pp. 29-39, September 1954.

[12] R. Pyndiah, Near-optimum decoding of product coder: block turbo codes, IEEE Transactions on Communication, vol. 46,no. 8, pp. 1003-1010, August 1998.

[13] E. Prange, *Cyclic Error-Correcting Codes in Two Symbols*, Air Force Cambridge Research Center, Cambridge, MA, September 1957.

[14] A. Hocquengbem, *Codes Correcteurs d'erreurs*, Chiffres, vol. 2, pp, 147-156, 1959.

[15] R. Bose and D. Ray-Chaudhuri, *On a Class of Error-Correcting Binary Codes*, Information and Control, vol. 3, pp. 68-79, 1960.

[16] I. Reed and G. Solomon, *Polynomial Codes over Certain Finite Fields*, Journal of the Society for Industrial and Applied Mathematics, vol. 8 , pp. 300-304, June 1960.

[17] R.G. Gallager, *Low-Density Parity-Check Codes*, IRE Transaction on Information Theory, vol. IT-8, pp. 21-28, January 1962

[18] G.D. Forney, *Concatenated Codes*, Cambridge, MA: MIT Press, 1966.

[19] D.J. Costello and G.D. Forney *Channel Coding: The Road to Channel Capacity*, Proceedings of the IEEE, 2006.

[20] C. Berrou, A. Glavieux, and P. Thitimajshima, *Near Shannon Limit Error- Correcting Coding and Decoding: Turbo-codes*, IEEE International Conference on Communications (ICC 1993), pp. 1064-1070, May 1993.

[21] J. Statman, J. Rabkin, and B. Siev, *Big Viterbi Decoder (BVD) Resultas for (7,1/2) Convolutional Code*, TDA Progress Report, vol. 42-99, pp. 122-129, November 1989.

[22] D.J. MacKay and R.M. Neal, *Good Codes Based on Very Sparse Matrices*, in Cryptography and Coding 5th IMA Conference, vol. 1025, pp. 100-1 11, 1995.

[23] G.E. Moore, *Codes and decoding on general graphs*, Electronics, vol. 38, no. 8, pp. 114–117, April, 1965.

[24] J.C. Carlach, *Contribution à la construction et au décodage à décision douce des codes correcteurs d'erreurs auto-duaux extrémaux*, Rapport de thèse, INSA de Rennes, 2012.

[25] G. Battail, M.C. Decouvelaere and P. Godlewski, *Replication Decoding*, IEEE Transaction On Information Theory, vol. IT-25, no. 3, May 1979.

[26] G. Battail et M. Decouvelaere, *Décodage par répliques*, Annales de Télécommunications, vol. 31, pp. 387-404, Novembre-Decembre. 1976.

[27] G. Battail et P. Godlewski, *Emploi de representations polynomiales à plusieurs indéterminées pour le décodage de codes redondants linéaires*, Annales de Télécommunications, vol. 33, 74-86, Mars-Avril. 1978.

[28] J.M. Wozencraft, *Sequential Decoding for Reliable Communication*, IRE National Convention Record, vol. 5, no. 2, pp. 11-25, August 1957.

[29] R.M. Fano, *a Heuristic Discussion of Probabilistic Decoding*, IEEE Transaction On Information Theory, vol. IT-9, pp. 64-74, April 1963.

[30] A.J. Viterbi, *Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*, IEEE Transaction on Information Theory, IT-13, pp. 260-269, April 1967.

[31] J.K. Omura, *On the Viterbi decoding algorithm*, IEEE Transaction on Information Theory, IT-15, pp. 177-179, May 1969.

[32] R.Bellman, *Dynamic Progranming*, Princeton University Press, 1957.

[33] G.D. Forney, *The Viterbi Algorithm*, Proceedings of the IEEE, vol. 61, no.3, pp. 268-278, March 1973.

[34] L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, *Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate* , IEEE Transactions on Information Theory, vol. IT-20, pp. 284-287, March 1974.

[35] J.K. Wolf, *Efficient Maximum Likelihood Decoding of Linear Block Codes Using a Trellis* , IEEE Transactions on Information Theory, vol. 20, pp. 76-80, June 1978.

[36] J.L. Massey, *Foundations and methods of channel encoding*, International Conference on Information Theory and Systems, pp. 148-157, 1978.

[37] G.D Forney, *Coset codes II: binary lattices and related codes*, IEEE Transactions on Information Theory, vol. 34, pp. 1152-1187, September 1988.

[38] S. Lin, L.G. Uehara, E. Nakamura, and W.P. Chu, *Circuit Design Approaches for Implementation of a Subtrellis IC for a Reed-Muller Subcode*, NASA Technical Report 96-001, February 1996.

[39] T. Fujiwara, H. Yamamoto, T. Kasami, S. Lin, *A trellis-based recursive maximum-likelihood decoding algorithm for binary linear block codes*, IEEE Transactions on Information Theory, Vol. 44, no. 2, pp. 714-729, 1998.

[40] Alexander Vardy, *Trellis structure of codes*, In V.S. Pless and W.C. Hukman, editors, Handbook of Coding Theory. Elsevier, 1998.

[41] A.R. Calderbank, G. D. Forney, A. Vardy, *Minimal Tail-Biting Trellises: The Golay Code and More*, IEEE Transactions on Information Theory, Vol. 45, no. 5, pp. 1435-1455, July 1999.

[42] R. Koetter and A. Vardy, *On the theory of linear trellis*, Information Coding and Mathematics, pp. 323–354, May 2002,

[43] R. Koetter and A. Vardy, *The structure of tail-biting trellises: minimality and basic principles*, IEEE Transactions on Information Theory, vol.49, pp. 2081-2105, September 2003.

[44] P. Shankar, P. N. Kumar, H. Singh and B.S Rajan, *Minimal tail-biting trellises for certain cyclic block codes are easy to construct*, International Colloquium on Automata, Languages and Programming (ICALP), pp. 627-638, 2001.

[45] T. K. Moon, *Error Correction Coding Mathematical Methods And Algorithms*, Wiley Interscience, 2005.

[46] J. Hagenauer and P. Hoeher, *A Viterbi Algorithm with Soft-Decision Outputs and its applications*, IEEE Global Telecommunications Conference, pp. 1680-1686, Novembre 1989.

[47] W. Koch and A. Baier, *Optimum and sub-optimum detection of coded data disturbed by time-varying inter-symbol interference*, IEEE Global Telecommunications Conference, pp. 1679–1684, Decembre 1990.

[48] P. Robertson, E. Villebrun, and P. Hoeher, *A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain*, IEEE International Conference on Communications, pp. 1009-1013, June 1995.

[49] N. Wiberg, *Codes and Decoding on General Graphs*, PhD. dissertation, Linköping University, Linköping, Sweden, 1996.

[50] R. M. Tanner, *A recursive approach to low complexity codes* , IEEE Transactions on Information Theory, vol. 27, pp. 533–547, September 1981.

[51] J.C. Carlach and C. Vervoux, *A New Family of Block Turbo-Codes*, Proceedings of the 13th International Symposium on Applied Algebra, Algebraic and Error-Correcting Codes(AAECC-13), pp. 15-16, 1999, Hawaii(USA).

[52] S. T. Brink, *Convergence of iterative decoding* , Electronics Letters, vol. 35 no. 10, May 1999.

[53] S. T. Brink, *Convergence of iterative decoding* , Electronics Letters, vol. 35 no. 13, June 1999.

[54] C. Berrou, M. Jezequel, C. Douillard, S. Kerouedan and L. Conde Canencia, *Duo Binary Turbo Codes Associated With High-Order Modulations* , ESA Workshop on Tracking Telemetry and Command Systems for Space Applications, Pays Bas, Octobre 2001.

[55] E. Arikan, *Channel polarization: a method for constructing capacity- achieving codes for symmetric binary-input memoryless channels*, IEEE Transaction on Information Theory, vol. 55, no. 7, pp. 3051–3073, 2009.

[56] B. M. Terhal, *Quantum Error Correction for Quantum Memories*, disponible sur le lien `http://arxiv.org/abs/1302.3428`, April 2015.

[57] F. R. Kschischang and V. Sorokine, *On the trellis structure of block codes* , IEEE Transactions on Information Theory, vol.41 pp.1924-1937, November 1995.

[58] T. Kasami, T. Takata, T. Fujiwara, and S. Lin, *On the optimum bit orders with respect to the state complexity of trellis diagrams for binary linear codes*, IEEE Transactions on Information Theory, vol. 39, no. 1, pp. 242-243, January 1993.

[59] O. Rioul, *Corps finis*, Télécom PariTech, Novembre 2011.

[60] T. Kasami, T. Takata, T. Fujiwara and S. Lin, *On Complexity of Trellis Structure of Linear Block Codes*, IEEE Transactions on Information Theory, vol. 39, pp. 1057-1064, May 1993.

[61] S. Lin, T. Kasami,T. Fujiwara and M. Fossorier, *Trellises and Trellis Based Decoding Algorithms for Linear Block Codes*, Kluwar Academic Publisher, 1998.

[62] M. Taskaldiran, C.S. Richard, and I. Kale, *Unified scaling factor approach for turbo decoding algorithms*, Springer Science, Wireless Technology, Lecture Notes in Electrical Engineering, pp. 203-2013, 2009.

[63] J. Vogt, and A. Finger, *Improving the Max-Log-MAP turbo decoder*, Electronic Letters, vol. 36, no. 23, pp. 1937–1939, November 2000.

[64] D.W. Yue, and H.H. Nguyen, *Unified scaling factor approach for turbo decoding algorithms*, IET Communications, vol. 4 iss. 8, pp. 905-9014, 2010.

[65] S. Benedetto, G. Montorsi, D. Divsalar, and F. Pollara, *Soft-output decoding algorithms in iterative decoding of turbo codes*, JPL TDA Progress Report, vol. 42-124, pp. 63–87, 1996.

[66] S. Benedetto, G. Montorsi, D. Divsalar, and F. Pollara, *Soft input soft output MAP module to decode parallel and serial concatenated codes*, TDA Progress Report vol. 42-127, pp.1–20, 1996

[67] G. Colavolpe, G. Ferrari, and R. Raheli, *Extrinsic Information in Iterative Decoding: A Unified View*, IEEE Communications on Communications, vol. 49, no. 12, pp. 2088-2094, December 2001.

[68] G. Olocco and J.P. Tillich, *Iterative decoding of a new family of block turbo-codes*, 2nd International Symposium on Turbo Codes and Related Topics, Brest(France), September 2000.

[69] A. Otmani, *Recherche des codes extrémaux sous forme de codes cortex*, Rapport de DEA, Faculté des sciences de Limoges, Juin 1999.

[70] A. Otmani, *Codes cortex et construction de codes auto-duaux optimaux*, Rapport de thèse, Faculté des sciences de Limoges, Septembre 2002.

[71] E. Cadic, *Construction de Turbo Codes courts possédant de bonnes propriétés de distance minimale*, Rapport de thèse, Faculté des sciences de Limoges, Septembre 2003.

[72] J.E. Perez-Chamorro, *Analog Decoding of the Cortex Codes*, PhD thesis, Sup-Telecom Bretagne/Université de Bretagne-Sud (France), 2009.

[73] J. Perez-Chamorro, C. Lahuec, F. Seguin, G. Le Mestre, and M. Jézéquel, *A Sub-threshold PMOS Analog Cortex Decoder for the (8, 4, 4) Hamming Code*, ETRI Journal, Volume 31, Number 5, October 2009.

[74] P. Adde, C. Jégo, R. L. Bidan and J. P. Chamorro, *Design and implementation of a soft-decision decoder for cortex codes*, 17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS), pp. 663–666, 2010.

[75] P. Adde, D.G. Toro and C. Jego, *Design of an Efficient Maximum Likelihood Soft*

Decoder for Systematic Short Block Codes, IEEE Transaction on Signal Processing, vol.60, No. 7, July 2012.

[76] D. Chase, *A Class of Algorithms for Decoding Block Codes With Channel Measurement Information*, IEEE Transaction on Information Theory, vol.IT-18, No. 1, January 1972.

[77] M. Arzel, C. Jégo, W. Gross, and Y. Bruned, *Stochastic multiple sream decoding of cortex codes*, IEEE Transactions on Signal Processing, pp. 3486–3491, 2011.

[78] P. Adde, *Vecteurs de test pour le décodage du code de Golay(24,12,8)*, Communication privée, 2013.

[79] J.W. Cooley and J.W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Computation , vol. 19, pp. 297-301, April 1965.

[80] J.C Carlach et S. Mohamed-Mahmoud, *Procédé et dispositif de décodage optimal des codes correcteurs d'erreurs linéaires*, brevet DI.200281, Janvier 2014.

[81] N. Seshadri and C.E. Sundberg, *List Viterbi decoding algorithms with applications*, IEEE Transactions on Communications, vol.42, no. 2/3/4, pp. 313–323, February–April 1994.

[82] M. Röder and R. Hamzaoui, *Fast Tree-Trellis List Viterbi Decoding*, IEEE Transactions on Communications, vol. 54, no. 3, March 2006

[83] S. Mohamed-Mahmoud, J.C Carlach, P. Adde et M. Jézéquel, *Décodage itératif des codes correcteurs d'erreurs courts en bloc linéaires basé sur des treillis produits sectionnalisés*, GRETSI'15, Septembre 2015.

[84] S. Lin, T. Kasami, T. Fujiwara, and M. Fossorier, *Trellis and Trellis-Based Decoding Algorithms for linear block codes*, Kluwar Academic Publishers, 1998.

[85] Y. Liu, S. Lin and M. Fossorier, *MAP Algorithme for Decoding Linear Block Codes Based on Sectionnalized Trellis Diagrams*, IEEE Transactions on Communications, vol. 48, pp. 21-28, April 2000.

[86] A. Lafourcade and A. Vardy, *Optimal Sectionalization of a Trellis*, IEEE Transactions on Information Theory, vol. 42, no. 3, pp. 689-703, May 1996.

Annexe A

CALCUL DES PROBABILITÉS α, β ET γ DANS L'ALGORITHME BCJR

Soit $\mathscr{C}(n,k)$ un code en bloc linéaire défini sur $\mathbb{F}_2$ et $\boldsymbol{c} = (c_0, c_1, \ldots, c_{n-1})$ un mot de code de $\mathscr{C}$. La modulation utilisée est la modulation BPSK qui associe le bit 0 (resp. 1) à la valeur modulée $-E_c$(resp $+E_c$), où E_c est l'enérgie transmise par bit de code. Le mot modulé $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ associé au mot c est ensuite transmis sur un canal à bruit blanc additif gaussien (AWGN) de variance $\sigma^2 = N_0/2$ avec fading dont l'amplitude est notée a. Le mot reçu est noté $\boldsymbol{y} = (y_0, y_1, \ldots, y_{n-1})$.

A.1 Calcul de la probabilité de branche γ_t

La probabilité de branche γ_t est donnée par:

$$\begin{aligned}\gamma_t(s_t, s_{t+1}) &= P(y_t, s_{t+1}|s_t) \\ &= P(y_t|s_t, s_{t+1})P(s_{t+1}|s_t)\end{aligned} \tag{A.1}$$

La probabilité $P(s_{t+1}|s_t)$ qu'un état s_{t+1} de l'étage $t+1$ soit atteint sachant l'état s_t de l'étage t est égale à la probabilité que le bit soit $c_t = 0,1$ corresponde à l'étiquette de branche connectant s_t à s_{t+1}. Donc $P(s_{t+1}|s_t) = P(c_t)$.
La transition (s_t, s_{t+1}) est équivalente à l'occurence du symbôle correspondant x_t, où x_t est la valeur modulée associée à la valeur du bit c_t. Donc $P(y_t|s_t, s_{t+1}) = P(y_t|x_t)$. La probabilité de branche est donc donnée par:

$$\gamma_t(s_t, s_{t+1}) = P(y_t|x_t)P(c_t) \tag{A.2}$$

La probabilité a priori $P(c_t)$, $c_t = 0,1$,peut être donnée par:

$$P(c_t = 1) = \frac{e^{L(c_t)}}{1 + e^{L(c_t)}} \tag{A.3}$$

$$P(c_t = 0) = \frac{1}{1 + e^{L(c_t)}} \tag{A.4}$$

Une expression générale de $P(c_t)$ est donnée par:

$$\begin{aligned} P(c_t) &= \frac{e^{\frac{(1+x_t)L(c_t)}{2}}}{1 + e^{L(c_t)}} \\ &= \frac{e^{\frac{L(c_t)}{2}}}{1 + e^{L(c_t)}} e^{\frac{x_t L(c_t)}{2}} \\ &= C_{1t} e^{\frac{x_t L(c_t)}{2}} \end{aligned} \tag{A.5}$$

où $L(c_t) = \log\left[\frac{P(c_t=1)}{P(c_t=0)}\right]$ est le LLR a priori de c_t et $C_{1t} = \frac{e^{\frac{L(c_t)}{2}}}{1+e^{L(c_t)}}$. C_{1t} ne dépend pas de la valeur du bit c_t.

La probabilité $P(y_t|x_t)$ dépend du canal et de la modulation utilisée. Pour un canal AWGN et une modulation BPSK, le signal à la sortie du filtre adapté est d'amplitude $y_t = a x_t \sqrt{E_c} + n$, où n est un bruit gaussien de moyenne nulle et de variance $\sigma_n^2 = N_0/2$. En normalisant l'amplitude du signal reçu on trouve:

$$\frac{y_t}{\sqrt{E_c}} = a x_t + \frac{n}{\sqrt{E_c}} = a x_t + n'. \tag{A.6}$$

où $n' = \frac{n}{\sqrt{E_c}}$ est un bruit gaussien de variance $\sigma_{n'}^2 = (\sigma_n/\sqrt{E_c})^2 = N_0/2E_c$. La probabilité $P(y_t|x_t)$ est donnée par:

$$\begin{aligned} P(y_t|x_t) &= \frac{1}{\sqrt{\pi N_0 E_c}} e^{-\frac{E_c}{N_0}(y_t - a x_t)^2} \\ &= \frac{1}{\sqrt{\pi N_0 E_c}} e^{-\frac{E_c}{N_0} y_t^2} e^{-\frac{E_c}{N_0} a^2 x_t^2} e^{\frac{2aE_c}{N_0} x_t y_t} \\ &= C_{2t} e^{\frac{L_c}{2} x_t y_t} \end{aligned} \tag{A.7}$$

où $C_{2t} = \frac{1}{\sqrt{\pi N_0 E_c}} e^{-\frac{E_c}{N_0} y_t^2} e^{-\frac{E_c}{N_0} a^2 x_t^2}$ est une quantité qui ne dépend pas de x_t car $x_t^2 = 1$ et $L_c = \frac{4aE_c}{N_0}$.

Nous remplaçons les probabilités $P(y_t|x_t)$ et $P(c_t)$ dans l'équation A.2 par leurs expressions, respectivement, dans les équations A.5 et A.7.

$$\gamma_t(s_t, s_{t+1}) = C_t e^{\frac{x_t L(c_t)}{2}} e^{\frac{L_c}{2} x_t y_t} \tag{A.8}$$

où $C_t = C_{1t} C_{2t}$.

A.2 Calcul de la probabilité forward α_t

En utilisant la théorie des probabilités on a $P(A) = \sum_B P(A,B)$, alors $\alpha_t(s_t)$ peut être exprimée comme suit:

$$\begin{aligned}\alpha_t(s_t) &= P(s_t, \boldsymbol{y}_0^{t-1}) \\ &= P(s_t, \boldsymbol{y}_0^{t-2}, y_{t-1}) \\ &= \sum_{s_{t-1}\in\sigma_{t-1}(s_t)} P(s_{t-1}, s_t, \boldsymbol{y}_0^{t-2}, y_{t-1}) \end{aligned} \tag{A.9}$$

Comme le canal est supposé sans mémoire et en utilisant la loi de Bayes, on a :

$$\begin{aligned}\sum_{s_{t-1}\in\sigma_{t-1}(s_t)} P(s_{t-1}, s_t, \boldsymbol{y}_0^{t-2}, y_{t-1}) &= \sum_{s_{t-1}\in\sigma_{t-1}(s_t)} P(s_t, y_{t-1}|s_{t-1}, \boldsymbol{y}_0^{t-2}) P(s_{t-1}, \boldsymbol{y}_0^{t-2}) \\ &= \sum_{s_{t-1}\in\sigma_{t-1}(s_t)} P(s_t, y_{t-1}|s_{t-1}) P(s_{t-1}, \boldsymbol{y}_0^{t-2}) \\ &= \sum_{s_{t-1}\in\sigma_{t-1}(s_t)} \gamma_t(s_{t-1}, s_t)\alpha_{t-1}(s_{t-1}) \end{aligned} \tag{A.10}$$

Donc,

$$\alpha_t(s_t) = \sum_{s_{t-1}\in\sigma_{t-1}(s_t)} \gamma_t(s_{t-1}, s_t).\alpha_{t-1}(s_{t-1}) \tag{A.11}$$

Pour éviter les problèmes d'instabilité numérique, il est recommandé de normaliser la valeur de α comme suit:

$$\alpha_t(s_t) = \frac{\alpha_t(s_t)}{\sum_{s_t}\alpha_t(s_t)} \tag{A.12}$$

A.3 Calcul de la probabilité backward β_t

L'expression récursive de β_t est obtenue de la même façon que α_t. Par définition $\beta_t(s_t) = P(\boldsymbol{y}_t^{n-1}|s_t)$. En utilisant l'expression $P(A) = \sum_B P(A,B)$ et la loi de Bayes alors:

$$\begin{aligned}
\beta_t(s_t) &= P(\boldsymbol{y}_t^{n-1}|s_t) \\
&= \sum_{s_{t+1}\in\sigma_{t+1}(s_t)} P(s_{t+1}, \boldsymbol{y}_t^{n-1}|s_t) \\
&= \sum_{s_{t+1}\in\sigma_{t+1}(s_t)} P(s_{t+1}, y_t, \boldsymbol{y}_{t+1}^{n-1}|s_t) \\
&= \sum_{s_{t+1}\in\sigma_{t+1}(s_t)} P(\boldsymbol{y}_{t+1}^{n-1}|s_t, s_{t+1}, y_t) P(s_{t+1}, y_t|s_t) \\
&= \sum_{s_{t+1}\in\sigma_{t+1}(s_t)} P(\boldsymbol{y}_{t+1}^{n-1}|s_{t+1}) P(s_{t+1}, y_t|s_t) \\
&= \sum_{s_{t+1}\in\sigma_{t+1}(s_t)} \beta_{t+1}(s_{t+1})\gamma_t(s_t, s_{t+1}) \qquad \text{(A.13)}
\end{aligned}$$

Pour éviter les problèmes d'instabilité numérique, il est recommandé de normaliser la valeur de β_t comme suit:

$$\beta_t(s_t) = \frac{\beta_t(s_t)}{\sum_{s_t}\beta_t(s_t)} \qquad \text{(A.14)}$$

Annexe B

DESCRIPTION DES DIFFÉRENTS BLOCS DANS L'ARCHITECTURE MATÉRIELLE DE L'ALGORITHME DE DÉCODAGE PAR TREILLIS-PRODUITS POUR LE CODE DE GOLAY$(24, 12, 8)$

B.1 Description des variables entrées-sorties des blocs

- $lanceCompt$: impulsion provenant de la machine d'état qui permet d'activer les compteurs.
- DM : impulsion provenant de la machine d'état qui permet, lorsqu'elle passe à 1, de remettre les compteurs à zéro au coup d'horloge suivant pour les relancer.
- H : horloge.
- $noSection$: compteur permettant de parcourir les sections du treillis-produit dans le sens forward ou backward de l'algorithme Max-Log-MAP.
- $compteur\alpha$: ce compteur permet d'identifier les valeurs stockées dans le bloc mémoire "Mémoire α". Si son bit de poids fort est mis à 0, la valeur de α à son entrée doit être stockée à l'emplacement $noSection$. Si son bit de poids fort est mis à 1, le bloc délivre à sa sortie les valeurs de α correspondant à la section $noSection$.
- $finTreillis$: impulsion indiquant la fin du treillis et permettant le passage à un autre treillis-produit.
- $noTreillis$: valeur du compteur 6 donnant le numéro du treillis-produit sur lequel les calculs sont effectués.
- $dernierTreillis$: actif à 1 lorsque le compteur des treillis-produits arrive à la fin de son comptage. Il indique donc que tous les treillis-produits sont traités et

permet aussi de changer d'état dans la machine d'états.

- $nouvelleIteration$: impulsion provenant de la machine d'état et permettant d'incrémenter le compteur des itérations.
- $noIteration$: valeur du compteur 8 donnant le numéro de l'itération en cours.
- $derniereIteration$: indique, lorsqu'il est à 1, que le compteur des itérations arrive à la fin de son comptage. Il permet aussi de changer d'état dans la machine d'états.
- $trame$: entrée du décodeur, message de 24 métriques à sauvegarder pour effectuer une nouvelle itération de l'algorithme.
- $saveMessage$: impulsion provenant de la machine d'état indiquant, lorsqu'elle est mise à 1, à la mémoire de mémoriser les informations entrantes.
- $symbole$: représente le symbole, dans la trame, correspondant au numéro de section $noSection$.
- α_{t-1}/β_{t+1} : si le bloc "Section t" effectue les calculs dans le sens forward, il reçoit à son entrée les valeurs des α_{t-1} calculées à l'étage $t-1$ du treillis-produit pour effectuer les calculs des α à l'étage t (α_t). Alors s'il effectue les calculs dans le sens backward, il reçoit à son entrée les valeurs des β_{t+1} calculées à l'étage $t+1$ du treillis-produit pour effectuer les calculs des β à l'étage t (β_t). Les 4 valeurs de α (ou β) sont représentées par 4 signaux de 12 bits chacun.
- γ_t : valeur de la métrique de branche γ à l'instant t.
- $sectionUtilisee$: détermine la structure de la section utilisée pour faire les calculs. Il y a 4 dispositions possibles de sections dans un treillis-produit à 4 états(voir paragraphe 3.3 du chapitre 3).
- α_0/β_n : mise à zéro de la mémoire sur l'indication de la machine d'état, ceci permet d'initialiser les métriques d'états de départ et de fin du treillis-produit ($\alpha_0 = 0$

et $\beta_n = 0$) au début des phases forward et backward respectivement.

- $calcul$: impulsion provenant de la machine d'état ordonnant le bloc "Calc/Comp LLR" à procéder au calcul du LLR de symbole correspondant à la section t du treillis-produit.
- LLR_t: le LLR du symbole correspondant à la section t du treillis-produit $noTreillis$.
- $motLLR$: le vecteur des LLRs des 24 symboles du mot de code émis.
- $dataSortie$: représente les 12 symboles de données utiles du mot trouvé.
- $MDCVrai$: booléen indiquant, lorsqu'il est à 1, si le mot trouvé est un mot de code.
- $message$: 12 symboles de données utiles.
- $melangeTrame$: permet d'indiquer, lorsqu'il est mis à 1, au bloc "mélange LLR_trame" d'additionner, symbole par symbole, les deux vecteurs à son entrée à savoir $trame$ et $motLLR$.
- $nouveauMessage$: nouvelle trame à sauvegarder dans le bloc mémoire "Mémoire trame" afin d'entamer une nouvelle phase de l'algorithme.

B.2 Description des différents blocs de la partie opérative

B.2.1 Bloc "Compteur 48"

Compte de 0 à 23 suivant l'horloge puis décompte de 55 à 32. Lorsque le compteur est en mode comptage, le bit de poids fort de $compteur\alpha$ est mis à 0, ceci indique que l'algorithme Max-Log-MAP est dans une phase forward pour le calcul des α et que les valeurs de ces dernières doivent être stockées dans la mémoire "Mémoire α". Lorsque le compteur est en mode décomptage, le bit de poids fort de $compteur\alpha$ est à 1, ceci

indique que l'algorithme Max-Log-MAP est dans une phase backward pour le calcul des β et que le bloc "Mémoire α" doit délivrer au bloc "Calc/Comp LLR" les valeurs de α correspondant à la section $noSection$. Lorsque le décomptage est fini, il y a une impulsion sur $finTreillis$ pour indiquer la fin de traitement du treillis-produit en cours. Le comptage recommence si $lanceCompt$ est actif.

- Entrées du bloc: $lanceCompt$, DM, H.
- Sorties du bloc: $noSection$, $compteur\alpha$, $finTreillis$.

B.2.2 Bloc "Compteur 6"

C'est un compteur des treillis-produits dont le nombre total est 6 treillis. Il compte donc de 0 à 5 suivant l'horloge et le signal $finTreillis$. Il est incrémenté tous les 48 coups d'horloge. La Sortie $dernierTreillis$ permet de changer d'état au niveau de la machine d'état.

- Entrées du bloc: $lanceCompt$, $finTreillis$, DM, H.
- Sorties du bloc: $noTreillis$, $dernierTreillis$.

B.2.3 Bloc "Compteur 8"

C'est un compteur des itérations qui sont éffectuées par l'algorithme. Le nombre maximal d'itérations est fixé à 8 itérations. Le compteur compte donc de 0 à 7 suivant l'horloge et le signal $nouvelleIteration$. Le compteur est incrémenté si le décodeur doit effectuer une nouvelle itération. La Sortie $derniereIteration$ permet de changer d'état au niveau de la machine d'état et de revenir au repos si aucun mot de code n'a été trouvé dans les 8 itérations.

- Entrées du bloc: $lanceCompt$, $nouvelleIteration$, DM, H.
- Sorties du bloc: $noIteration$, $derniereIteration$.

B.2.4 Bloc "Mémoire trame"

Mémorise la trame de 24 symboles en entrée du décodeur lorsque le signal $sauveMessage$ est active. L'entrée $noSection$ permet de sélectionner l'emplacement où il faut sauvegarder le $symbole$. La sortie du bloc notée $trame$ sera utilisée lors des itérations en l'additionnant avec le vecteur des LLRs des symboles $motLLR$ calculé dans l'itération précédente.

- Entrées du bloc: $symbole$, $saveMessage$, $noSection$.
- Sorties du bloc: $trame$.

B.2.5 Bloc "Section t"

Ce bloc effectue le clcul des α (respectivement β) dans le sens forward (respectivement backward) sur une section donnée du treillis-produit. Il contient les informations liées aux treillis-produits ainsi que les 4 cas possibles pour la représentation d'une section. Ce bloc est divisé en sous blocs :

- Une mémoire pour les informations des différents treillis.
- Un bloc représentant chaque section possible et permettant d'effectuer les calculs correspondants à celle-ci (4 blocs en tout).

Les entrées/sorties du bloc sont:

- Entrées du bloc: $symbole$, $noSection$, $noTreillis$, α_{t-1}/β_{t+1}.
- Sorties du bloc: α_t/β_t, γ_t, $sectionUtilisee$.

B.2.6 Bloc "Mémoire α/β"

Sauvegarde les 4 valeurs de α (respectivement β) calculées à un instant donné du treillis dans la phase forward (respectivement backward) afin de les réutiliser dans le calcul

des α (respectivement β) à l'instant suivant. Lorsque le calcul sur un nouveau treillis commence ou que le calcul des β commence, la machine d'état réinitialise la mémoire à zéros grâce au signal d'entrée α_0/β_n.

- Entrées du bloc: α_0/β_n, α_t/β_t.
- Sorties du bloc: α_{t-1}/β_{t+1}.

B.2.7 Bloc "Mémoire α"

Lorsque $compteur\alpha$ est actif à 0, le bloc mémorise les informations de la section dans les cases correspondantes à $noSection$. Lorsque $compteur\alpha$ est inactif à 1, le bloc délivre les valeurs des α associées à $noSection$.

- Entrées du bloc: α_{t-1}, $compteur\alpha$, $noSection$.
- Sorties du bloc: α_{t-1}.

B.2.8 Bloc "Calc/Comp LLR"

Grace aux trois informations en entrée, ce bloc calcule les 4 LLR possibles de la section du treillis désignée par SectionUtilisee selon l'équation $LLR_t = \alpha_{t-1}+\beta_t+\gamma_t$ On compare ensuite les quatre résultats et on choisit le meilleur qui est mis en sortie du bloc.

- Entrées du bloc: α_{t-1}, β_t, $sectionUtilisee$, $calcul$.
- Sorties du bloc: LLR_t.

B.2.9 Bloc "Mémoire LLR"

Sauvegarde le LLR pour chaque section de chaque treillis. Restitue l'ensemble des LLR d'un treillis (24 symboles) lorsque la ligne de la mémoire correspondant à ce treillis est complète.

- Entrées du bloc: LLR_t, $noSection$, $noTreillis$.
- Sorties du bloc: $motLLR$.

B.3 Bloc "Codage/Comp MDC"

Ce bloc récupère le mot trouvé en sortie de la mémoire des LLR (soit 24 symboles). Il sépare les 12 symboles data des 12 symboles de parité. La data est récupéré puis recodé à l'aide de la matrice génératrice : un multiplexeur 12 vers 7 permet de choisir les datas utilisées pour le calcul du code. Les valeurs des parités sont calculées successivement (il y en a 12) et stockées dans une mémoire. Les informations de parité créées sont ensuite comparées à celles du mot trouvé. Si ce sont les mêmes, alors le mot trouvé est un mot de code. MDC_vrai est alors activé à 1. Les symboles data du mot trouvé sont systématiquement envoyés sur la sortie Data_sortie. Ce bloc est composé de sous blocs :

- Un bloc séparant les symboles de parité de ceux data
- Un bloc de calcul pour le nouveau mot de code (composé d'un multiplexeur, d'un compteur12 d'additionneurs et d'une mémoire).
- Et un comparateur pour les parités calculées et récupérées précédemment.
- Entrées du bloc: $motLLR$.
- Sorties du bloc: $dataSortie$, $MDCVrai$.

B.3.1 Bloc "Mémoire message"

Lorsque $MDCVrai$ est actif à 1, la mémoire sauvegarde les informations provenant de $dataSortie$ à l'emplacement $noTreillis$.

- Entrées du bloc: $dataSortie$, $MDCVrai$, $noTreillis$.

- Sorties du bloc: $message$.

B.3.2 Bloc "mélange LLR_trame"

Lorsqu'une nouvelle itération a lieu, $melangeTrame$ est activé à 1. $trame$ et $motLLR$ sont alors additionnés. Le résultat est ensuite tronqué pour ne faire que 8 bits et est mis sur la sortie $nouveauMessage$ (qui sera redirigé vers la mémoire du message à décoder pour pouvoir recommencer un nouveau cycle).

- Entrées du bloc: $trame$, $motLLR$, $melangeTrame$.
- Sorties du bloc: $nouveauMessage$.

Résumé

Pour des trames de longueurs n supérieures ou égales à quelques milliers de bits (n > 1000), les turbocodes et les codes LDPC (Low-Density Parity Check) ou leurs variantes donnent d'excellentes performances avec une complexité raisonnable pour des implémentations "temps réel" même pour des débits très élevés (GigaBit/s). Par contre pour les petites longueurs (n < 1000) de trames, les turbocodes et codes LDPC courts sont moins performants. En effet, pour avoir de grandes distances minimales dmin avec des codes de petites longueurs n et de grandes capacités de correction, il faut des codes dont les matrices de contrôle de parité soient de densité élevée et non creuse, c'est à dire avec beaucoup de 1 et très peu de 0, comme celles des codes LDPC ou des turbocodes. La densité élevée de la matrice de contrôle du code conduit à des cycles courts dans le graphe de Tanner correspondant et donc rend les algorithmes type BP (Belief Propagation) inefficaces.

L'objectif principal de cette thèse est donc de contribuer à améliorer le décodage à décision douce des codes en bloc linéaires courts. Nos travaux se sont orientés, dans un premier temps, vers l'étude d'une classe particulière de codes en bloc appelés codes Cortex. La construction Cortex repose sur l'utilisation de codes de base de petites longueurs assemblés en étages reliés par des permutations. Nous proposons une nouvelle méthode pour estimer les variables cachées dans la structure afin de pouvoir effectuer un décodage itératif entre les codes de base. Nous proposons une deuxième méthode de décodage qui ne nécessite pas d'estimation des variables cachées en les dissimulant dans des équations booléennes.

Dans un second temps nous avons élargi nos recherches à des algorithmes de décodage très généraux valables pour tout code en bloc. Nous proposons dans ce cadre trois méthodes de décodage basées sur des treillis-produits de complexité réduite construits à partir des lignes des matrices génératrice et de contrôle du code en bloc.

Mots-clés : Codes en bloc, Codes Cortex, Décodage à décision douce, Treillis, Décodage itératif, BCJR, SOVA, Viterbi

Abstract

For lengths n greater or equal to several thousands of bits, turbocodes and LDPC (Low-Density Parity Check) codes or theirs alternatives achieve a very good performance with reasonable implementation complexity in a real time even for very high data rates (GigaBit/s). However for small code lengths (n < 1000), turbocodes and LDPC codes are less efficient. Indeed, to get short block codes with high minimal distance dmin and high correction capabilities, theirs parity-check matrices must be not sparse i.e with much 1 and little 0 like the LDPC or turbocodes matrices. The high density of the parity check matrix yield to short cycles in the associated Tanner graph and therefore makes the algorithms type-BP (Belief Propagation) not efficient.

The main objective of this thesis is then to contribute to the soft decision decoding of short linear block codes. Our work was directed, initially, toward the study of a particular class of block codes called Cortex codes. Cortex construction is based on the use of component codes with small lengths assembled in stages connected by permutations. We propose a new method to estimate the hidden variables in the structure in order to perform iterative decoding between the component codes. We propose a second decoding algorithm which does not require estimation of hidden variables by concealing them in Boolean equations.

Secondly we have expanded our research to propose very general decoding algorithms valid for any code block. We propose in this context three decoding methods based on reduced complexity trellis-products constructed from the lines of generator and parity-check matrices of block code.

Keywords : Block codes, Cortex codes, Soft decoding, Trellis, iterative decoding, BCJR, SOVA, Viterbi

TELECOM Bretagne

n° d'ordre : 2016telb0394

Télécom Bretagne

Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3

Tél : + 33(0) 29 00 11 11 - Fax : + 33(0) 29 00 10 00